Test-driven Fault Navigation for Debugging Reproducible Failures

Dissertation
zur Erlangung des akademischen Grades eines
Doktors der Ingenieurwissenschaften
(Dr.-Ing.)
in der Wissenschaftsdisziplin „praktische Informatik“

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Universität Potsdam

von
Michael Perscheid, M.Sc.

Potsdam, den 25. September 2013

Betreuer:	Prof. Dr. Robert Hirschfeld Fachgebiet Software-Architekturen Hasso-Plattner-Institut für Softwaresystemtechnik Universität Potsdam	
Gutachter:	Prof. Dr. Stephan Diehl Lehrstuhl für Softwaretechnik FB IV Informatik Universität Trier	Prof. Dr. Oscar Nierstrasz Software Composition Group Institut für Informatik Universität Bern

“Debugging is the dirty little secret of computer science.”

— Henry Lieberman.

The Debugging Scandal and What to Do About It. 1997.

Abstract

The correction of software failures tends to be very cost-intensive because their debugging is an often time-consuming development activity. During this activity, developers largely attempt to understand what causes failures: Starting with a test case that reproduces the observable failure they have to follow failure causes on the infection chain back to the root cause (defect). This idealized procedure requires deep knowledge of the system and its behavior because failures and defects can be far apart from each other. Unfortunately, common debugging tools are inadequate for systematically investigating such infection chains in detail. Thus, developers have to rely primarily on their intuition and the localization of failure causes is not time-efficient. To prevent debugging by disorganized trial and error, experienced developers apply the *scientific method* and its systematic hypothesis-testing. However, even when using the scientific method, the search for failure causes can still be a laborious task. First, lacking expertise about the system makes it hard to understand incorrect behavior and to create reasonable hypotheses. Second, contemporary debugging approaches provide no or only partial support for the scientific method.

In this dissertation, we present *test-driven fault navigation* as a debugging guide for localizing reproducible failures with the scientific method. 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 localizes suspicious system parts and restricts the initial search space. Second, team navigation recommends experienced developers for helping with failures. Third, behavior navigation allows developers to follow emphasized infection chains back to root causes. Fourth, state navigation identifies corrupted state and reveals parts of the infection chain automatically. We implement test-driven fault navigation in our *Path Tools framework* for the Squeak/Smalltalk development environment and limit its computation cost with the help of our *incremental dynamic analysis.* This lightweight dynamic analysis ensures an immediate debugging experience 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 state properties are infected.

Zusammenfassung

Die Beseitigung von Softwarefehlern kann sehr kostenintensiv sein, da die Suche nach der Fehlerursache meist sehr lange dauert. Während der Fehlersuche versuchen Entwickler vor allem die Ursache für den Fehler zu verstehen: Angefangen mit einem Testfall, welcher den sichtbaren Fehler reproduziert, folgen sie den Fehlerursachen entlang der Infektionskette bis hin zum ursprünglichen Defekt. Dieses idealisierte Vorgehen benötigt ein grundlegendes Verständnis über das Systemverhalten, da Fehler und Defekt sehr weit auseinander liegen können. Bedauerlicherweise bieten jedoch gebräuchliche Entwicklungswerkzeuge wenig Unterstützung, um solche Infektionsketten detailliert zu untersuchen. Dementsprechend müssen Entwickler primär auf ihr Gespür vertrauen, so dass die Lokalisierung von Fehlerursachen sehr viel Zeit in Anspruch nehmen kann. Um ein willkürliches Vorgehen zu verhindern, verwenden erfahrene Entwickler deshalb die *wissenschaftliche Methode*, um systematisch Hypothesen über Fehlerursachen zu prüfen. Jedoch kann auch noch mittels der wissenschaftlichen Methode die Suche nach Fehlerursachen sehr mühsam sein, da passende Hypothesen meist manuell und ohne die systematische Hilfe von Werkzeugen aufgestellt werden müssen.

Diese Dissertation präsentiert die *test-getriebene Fehlernavigation* als einen zusammenhängenden Wegweiser zur Beseitigung von reproduzierbaren Fehlern mit Hilfe der wissenschaftlichen Methode. Basierend auf der Analyse von funktionierenden und fehlschlagenden Testfällen werden Anomalien aufgedeckt und in eine Breitensuche integriert, um Entwickler zum Defekt zu führen. Diese systematische Suche besteht aus vier spezifischen Navigationstechniken, welche zusammen die Erstellung, Evaluierung und Verfeinerung von Hypothesen für die wissenschaftliche Methode unterstützen. Erstens grenzt die Strukturnavigation verdächtige Systemteile und den initialen Suchraum ein. Zweitens empfiehlt die Team-Navigation erfahrene Entwickler zur Behebung von Fehlern. Drittens erlaubt es die Verhaltensnavigation Entwicklern, die hervorgehobene Infektionskette eines fehlschlagenden Testfalls zurückzuverfolgen. Viertens identifiziert die Zustandsnavigation fehlerhafte Zustände, um automatisch Teile der Infektionskette offenzulegen. Alle vier Navigationen wurden innerhalb des *Path Tools Framework* für die Squeak/Smalltalk Entwicklungsumgebung implementiert. Dabei bauen alle Werkzeuge auf die *inkrementelle dynamische Analyse*, welche die Berechnungskosten über mehrere Testdurchläufe abhängig von den Bedürfnissen des Nutzers aufteilt und somit schnelle Ergebnisse während der Fehlersuche liefert. Folglich können wichtige Fragen in kurzer Zeit beantwortet werden: Wo wird mit der Fehlersuche begonnen? Wer versteht Fehlerursachen am Besten? Was passierte bevor der Fehler auftrat? Welche Programmzustände sind betroffen?

Acknowledgements

This work would not have been possible without the great help of several people. I am deeply grateful to the following people who supported me during the endless time of writing this thesis:

Committee

Prof. Dr. Stephan Diehl, Prof. Dr. Robert Hirschfeld, Prof. Dr. Oscar Nierstrasz

Colleagues

Dr. Malte Appeltauer, Dr. Damien Cassou, Tim Felgentreff, Felix Geller, Dr. Michael Haupt, Robert Krahn, Jens Lincke, Tobias Pape, Bastian Steinert, Marcel Taeumel, Lauritz Thamsen, Sabine Wagner, Marcel Weiher

Collaborators

Design by Contract. Leonhard Schweizer, Oliver Richter
Mutation Engine. Anton Gulenko
PathFinder-Y. Matthias Jacob, Christian Kieschnick, Oliver Richter, Stefan Schaefer, Leonhard Schweizer
PathView. Franz Becker, Tim Felgentreff, Anton Gulenko, Markus Güntert, Stephanie Platz, Philipp Tessenow
Replay-driven Fault Navigation. Tim Felgentreff
Step-wise Run-time Analysis. Martin Beck, Felix Geller, Bastian Steinert
Test-driven Fault Navigation. Fabian Beck, Prof. Dr. Hidehiko Masuhara
Type Harvester. Dr. Michael Haupt

All professors and colleagues of the HPI research school

Family

Cindy Fähnrich, Heike Perscheid, Peter Perscheid, Petra Perscheid, Ruth Rißmann

Thank you!

Selbstständigkeitserklärung

Erklärung: Hiermit versichere ich, die vorliegende Dissertation selbstständig verfasst und durchgeführt zu haben. Es wurden keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, sind kenntlich gemacht. Die Arbeit war in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung.

Ich versichere weiterhin, dass ich diese Arbeit oder verwandte Abhandlungen nicht bei einer anderen Fakultät oder Universität eingereicht habe.

Potsdam, am 25. September 2013

(Michael Perscheid)

Contents

Part I.

Introduction and Background

1 Introduction

Software failures are ubiquitous and their consequences tend to be very cost-intensive. Due to the steadily growing complexity of software and time pressure during its development, almost every program contains failures. A large number of reports confirm that even professional applications are rarely free from errors [67, 136, 215, 219]. For example, the Ariane 5 launch vehicle crashed because of a simple type-conversion error and so caused damage of about $ 370 million [60]. Software failures are expensive for both users and software companies. In 2002, it was reported that software failures cost the U.S. economy about $ 60 billion per year [185]. Regarding the costs for software companies, a survey from 2008 found that the annual costs for solving software defects amount to $ 52,000 per developer [23]. In other words, a company with 100 developers pays about $ 5 million per year just for the effects of software failures.

Apart from malfunctions in business critical software systems, one major reason for the high costs of software failures is that debugging them is an extremely time-consuming development activity. Debugging as the reproduction, identification, and correction of failure causes is not trivial [67]; developers have to spend significant time to search for failure causes and to fix defects [23]. They require between 37 % [23] and 50 % [29] time solely to debugging activities. Companies report that up to 25 failures per year are so hard to debug that they need multiple developers and several workdays for correction [23]. Consequently, testing, debugging, and verification activities can easily range from 50 % to 75 % of the total development costs [97].

Software companies are gradually recognizing that their debugging concepts are inefficient and, therefore improvements in debugging processes could provide significant cost-savings. It is reported that 100 out of 139 North American companies consider their debugging practices to be problematic [23]. Although many of them apply code reviews, static analysis, and dynamic analysis tools, they often do not recognize defects and failure causes. In particular, time pressure and inadequate tool support makes debugging a challenging task. Developers and software organizations estimate that improvements in testing and debugging could cut total development costs by a third [23, 185]. In order to reduce the required time, effort, and cost of debugging, we argue that there is a need for investigating new approaches for correcting defects.

1.1. The Laborious Search for Failure Causes

Debugging is largely an attempt to understand what causes failures [219]. Starting with reproducing the observable failure in form of a test case, developers follow failure causes and their effects on the infection chain back to the root cause (defect). To localize failure causes, they examine involved program entities and distinguish relevant from irrelevant behavior and clean from infected state. After understanding all details of failure causes and their effects, developers are able to identify and correct the root cause. However, this idealized procedure requires deep knowledge of the system and its behavior [209] because failures and defects can be far apart from each other [137]. Developers may not be familiar with the numerous program entities that are involved in long-running infection chains. This forces them to understand source code they are less familiar with and to determine whether the behavior is correct or not.

Localizing failure causes tends to be a tedious activity because developers are not able to investigate infection chains in a systematic way. There are two main reasons: deficiencies of standard development tools and inadequate knowledge about debugging methods [90, 97, 150]. Common debugging tools—including symbolic debuggers and test runners—do not support identification and tracking of infection chains [140]. Symbolic debuggers only provide access to the last point of execution without access to the program history or hints about suspicious behavior. Similarly, test runners only verify that observable failures still exist. Both tools suffer from missing advice about causes and lack the capability to systematically follow failures back to their defects. Thus, developers have to work out on their own what is wrong and how to trace failure causes that occurred in the past. Debugging with these outdated but still prevalent tools leads more to a guessing game than a systematic procedure to localize failure causes. In addition to this, lacking debugging knowledge also forces developers to apply disorganized trial and error approaches [90, 150]. Programming courses often pay little attention to debugging and specific debugging courses are rare and optional. Therefore, developers have to rely exclusively on their experience and intuition. Without feedback about inefficient methods, this often leads to large differences in debugging skills. Experienced developers are able to locate defects up to three times faster and add fewer new failures than novices [88, 93].

Since common debugging tools are inadequate and developers rely primarily on their intuition, the localization of failure causes is a laborious and time-consuming activity. We summarize our problem statement with the still valid quotation:

Problem statement

> "When something does go wrong, the people who write programs still lack good ways of figuring out exactly what went wrong. Debugging is still, as it was 30 years ago, largely a matter of trial and error."
>
> — Henry Lieberman, 1997 [138].

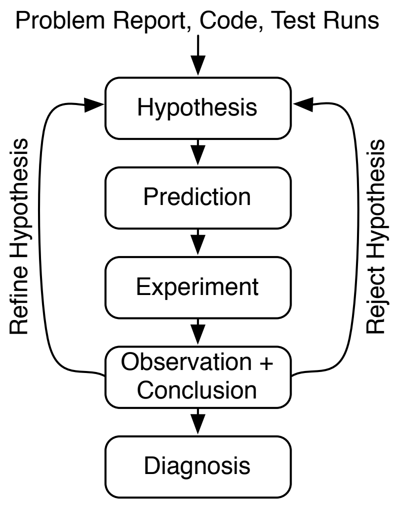


Figure 1.1.: The scientific method supports a systematic procedure to debug infection chains from observable failures back to defects.

To prevent debugging by disorganized trial and error, experienced developers apply the *scientific method* and its systematic hypothesis-testing [219, 150]. With the help of the scientific method as shown in Figure 1.1, developers are able to obtain thorough explanations for failures by creating, evaluating, and refining failure cause hypotheses. Starting with the observation of a reported failure, developers create an initial hypothesis concerning the problem. With this hypothesis, they predict program behavior and validate their expectations by experimenting with the system under observation. Depending on these experimental results, developers draw a conclusion that leads to a refined or alternative hypothesis. If this hypothesis is not able to explain earlier and predict future observations, developers repeat the prediction, experiment, and conclusion step. In the end, a conclusive hypothesis represents a diagnosis that determines the infection chain and the root cause of the failure. As the scientific method is a general process, there are a number of concrete debugging strategies that convert a hypothesis into a diagnosis [150]. Apart from strategies such as binary search, depth-first search, and deductive-analysis, breadth-first search has proven itself as the most efficient for experts [209]. In this case, developers start with a broad hypothesis about infected system parts that is increasingly refined with detailed program comprehension until the root cause is found.

However, even when using the scientific method the search for failure causes can still be a laborious task. In each iteration, developers have to choose from the countless possibilities of failure causes to create and evaluate suitable hypotheses. The efficiency of this procedure strongly depends on available expert knowledge and the abilities of debugging tools.

Lacking expertise about the system makes it hard to understand incorrect behavior and to create reasonable hypotheses [162]. Thus, inexperienced developers require more iterations, additional effort to conduct experiments, and a vast amount of time to find failure causes. Even if more experienced developers may help with debugging specific failures, their identification is challenging [11]. As infection chains and root causes are still unknown, it is hard to find expert knowledge for debugging new failures. For these reasons, assigned developers mostly have to debug on their own, independent of available program comprehension about failure causes and effects.

In addition to lacking expertise, contemporary debugging tools provide no or only partial support for the scientific method. For example, the prevalent symbolic debugger does not support developers either in creating hypotheses or in observing the execution history that leads to root causes. Furthermore, more state-of-the-art approaches do not focus on the entire scientific method. For example, spectrum-based fault localization [122] and likely invariant detection [68] reveal anomalies that may help in creating failure cause hypotheses. By comparing coverage and state of passed and failed test cases, such approaches produce prioritized lists of suspicious statements which restrict the search space and reduce guessing. Unfortunately, anomalies are not defects—developers have only numerous, unrelated starting points that must tediously be debugged one by one. Another example of current debugging approaches are back-in-time or omniscient debuggers [135]. These tools provide full access to past events so that developers can directly experiment with the entire infection chain. They are able to observe everything between the failure and its root cause. But back-in-time debuggers produce too much data which bear little relation to failure causes and their recording of program behavior comes with an expensive performance overhead. Thus, developers have no suggestions for creating failure cause hypotheses and experimentation tends to be slow.

Currently, there are no debugging methodologies that support the scientific method in an efficient way. For that reason, we summarize our research question as follows:

Research question

> How can we efficiently support developers in creating, evaluating, and refining failure cause hypotheses so that we reduce time and effort required for debugging?

We argue that the costs of debugging can be lower with better support for using the scientific method. With the help of an integral debugging approach and its corresponding tools, developers can systematically follow failures back to their root causes. Starting with a breadth-first search, they understand failure causes step by step from infected system parts to incorrect object states. Supported by integrated debugging tools that guide both novices and experts along infection chains, developers are able to create, evaluate, and refine failure cause hypotheses with less effort. Hence, such a systematic approach limits the influence of disorganized trial and error debugging and focuses developers' attention on finding defects. So, developers localize failure causes more efficiently with the result that they require less time and effort for the hardest part of debugging [90, 125, 150].

1.2. Our Approach

In this dissertation, we leverage test cases and analyze their hidden knowledge to propose a new and systematic debugging method that efficiently guides developers to the root causes of reproducible failures. Figure 1.2 summarizes our approach. Based on the idea of debugging into examples which proposes new perspectives on test cases, our test-driven fault navigation systematically leads developers to defects by combining the scientific method with anomalous behavior and state. While debugging with this approach, our incremental dynamic analysis efficiently collects and immediately presents the required run-time data. Finally, our Path Tools framework implements both test-driven fault navigation and incremental dynamic analysis for the Squeak development environment.

Debugging into examples provides new perspectives on test cases that form the basis of our systematic debugging approach. We consider not only passing and failing test case results, but also their deterministic and reproducible system behavior. Test cases are small examples that describe how the system works or not. Assuming that test cases provide entry points into reproducible system behavior, we are able to observe and analyze their execution paths step by step. Thus, we provide access to execution histories of reproducible failures instead of only their resulting error messages. So, developers are able to understand entire infection chains of failing test cases.

Apart from the new perspective on one specific test case, we also consider the relationship between several test cases and reveal their hidden knowledge. Test cases possess a valuable source of information as they implicitly define expected and unexpected behavior all over the system. During the execution of their exemplary assertions, they do not merely cover directly-tested methods but rather large call sequences through internal parts of the system. With an entire analysis and comparison of all behavior and state details on these execution paths, we are able to expose several differences in passing and failing runs. Such distinctions reveal anomalies that are valuable indications of failure causes. In combination with the execution history of a failing test case, anomalies are able to emphasize the infection chain and to guide developers along failure causes and their effects.

Test-driven fault navigation is a debugging guide that integrates anomaly detection into a breadth-first search for systematically creating, evaluating, and refining failure cause hypotheses. Based on the scientific method and our new perspectives on test cases, we define a novel and systematic debugging method that applies the hidden test knowledge to support developers in localizing failure causes and understanding how failures come to be. While the scientific method in the form of a breadth-first search provides a framework for narrowing down failure causes, our anomalies automatically emphasize failure causes within structure, behavior, and state of programs. Developers start by creating hypotheses about suspicious system parts (structure navigation). Optionally, if developers need more help, we recommend experts for proposing hypotheses and explaining causes (team navigation). After that, suitable developers make predictions about anomalous behavior

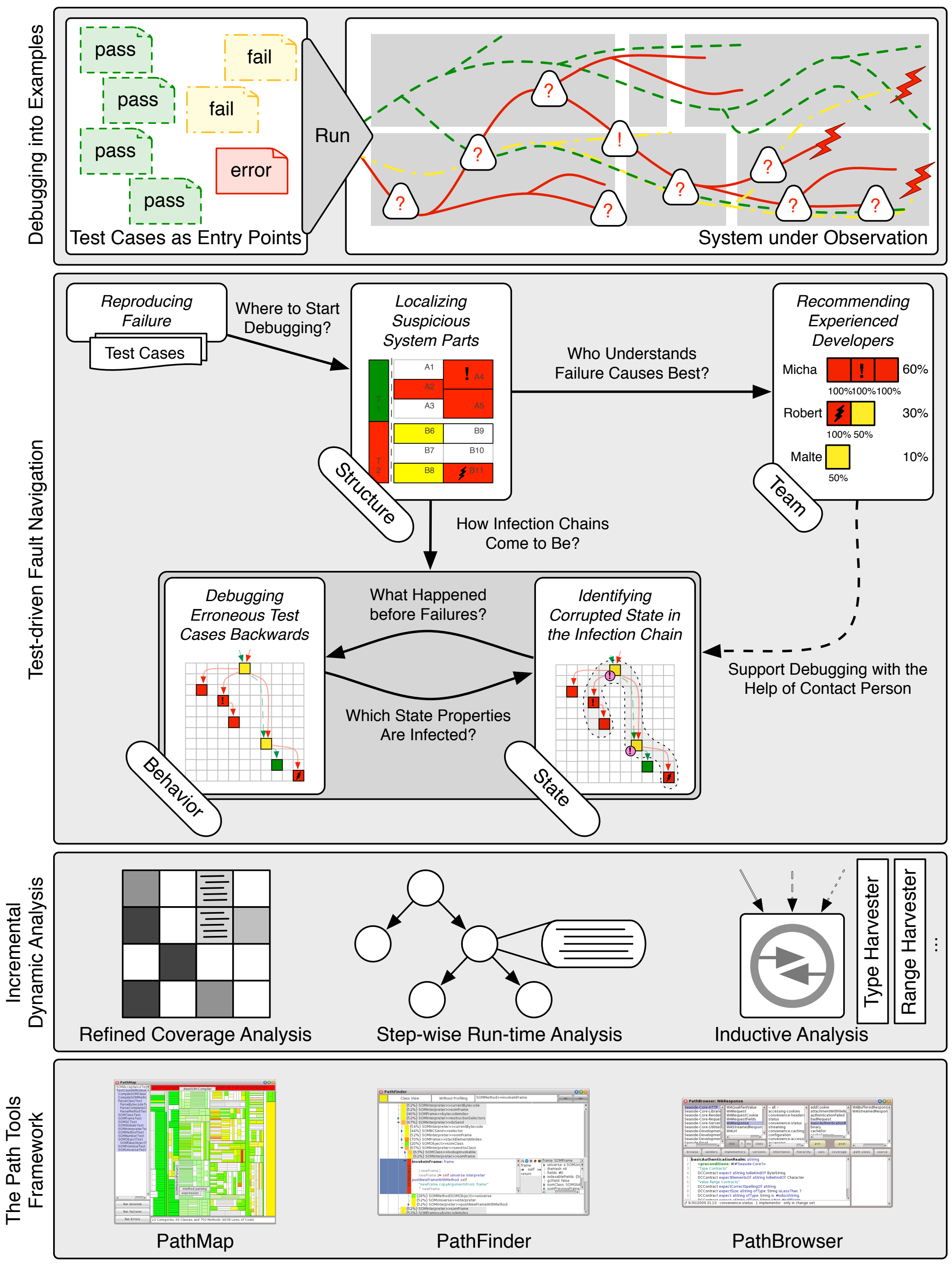

Debugging into Examples
pass
pass
pass
pass
fail
fail
error
Test Cases as Entry Points
Run
System under Observation
Test-driven Fault Navigation
Reproducing Failure
Test Cases
Where to Start Debugging?
Localizing Suspicious System Parts
A1
A2
A3
A4
A5
B6
B7
B8
B9
B10
B11
Structure
Who Understands Failure 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 State Properties Are Infected?
Identifying Corrupted State in the Infection Chain
Behavior
State
Support Debugging with the Help of Contact Person
Incremental Dynamic Analysis
Refined Coverage Analysis
Step-wise Run-time Analysis
Inductive Analysis
Type Harvester
Range Harvester
...
The Path Tools Framework
PathMap
PathFinder
PathBrowser

Figure 1.2.: An overview of our approach.

and experiment with the execution history of a failing test case (behavior navigation). Finally, our anomalies reveal the infection chain and assist developers in the observation of and conclusion about failure causes (state navigation).

Incremental dynamic analysis is a novel and lightweight dynamic analysis technique that ensures an immediate experience when debugging with test-driven fault navigation. Our navigation techniques require fast access to a wide variety of run-time information in order to offer developers the requested guidance to failure causes as soon as possible. However, applicable dynamic analysis tools are often associated with an inconvenient overhead because they impose time-consuming in-depth analyses and expensive setups. These issues render contemporary tools rather impractical for instant access to debugging data [201]. For that reason, we employ a new approach to dynamic analysis that achieves fast response times for requested run-time information that current tools are missing. Our analysis ensures a high degree of scalability by automatically splitting and distributing the dynamic analysis across multiple runs depending on the developers' needs. We propose an interactive approach to collect and present run-time data that leverages test cases as reproducible entry points into the execution of programs. First, an initial analysis provides immediate access to visualization of shallow run-time information. Second, as users explore this information, it is incrementally refined on demand by executing and analyzing corresponding test cases again. Our incremental dynamic analysis yields the low response times necessary for an immediate debugging experience [201] while preserving the quality of analysis results. Thus, we allow for a practical, spontaneous, and efficient use of our test-driven fault navigation in order to reduce the required debugging time further.

The Path Tools framework implements our test-driven fault navigation and incremental dynamic analysis in the Squeak development environment [112], an implementation of the Smalltalk programming language [87]. Our implementation consists of three tools that are built on top of our flexible dynamic analysis framework. *PathMap* is an extended test runner for supporting structure, team, and state navigation. It visualizes suspicious system parts and identifies experienced developers for helping with failures. *PathFinder* is our lightweight back-in-time debugger for navigating through specific test case behavior and state. It assists developers in localizing root causes by accessing entire execution histories, highlighting infection chains, and answering questions about object states. *PathBrowser* connects the hidden test knowledge with a source code editor to further support developers in program comprehension. The *Path analysis framework* provides the basis for our tools. It is an extensible realization of our incremental dynamic analysis for Smalltalk's SUnit framework[1]. By leveraging unit tests as a basis for dynamic analysis, we can ensure reproducibility and a high degree of automation, scalability, and performance during debugging with our tools.

[1]We consider unit test frameworks, for example xUnit [27], as a technique for implementing different kinds of test cases. Our approach works for combined testing including among others acceptance, integration, and module tests.

To evaluate our test-driven fault navigation, incremental dynamic analysis, and Path Tools framework, we conduct several experiments to show its practicality, effectiveness, and efficiency for debugging reproducible failures. First, we observe and compare developers while debugging with common debugging tools and our Path Tools. In this user study, we discover that test-driven fault navigation is not only able to decrease the total debugging time, but also to lower the time differences between individual developers. The feedback of our participants acknowledges that it was easier for them to find proper hypotheses and to follow failures back to root causes. Second, we assess the effectiveness of our test-driven heuristics for recommending experienced developers and revealing infection chains. Our results are promising with respect to restricting the search space of several mutated applications Even if root causes are still unknown, we propose suitable developers and valuable hints for failure causes. Finally, we measure the efficiency of incremental dynamic analysis and our Path Tools. We can ensure an experience of immediacy with our tools because our analysis implies only a low performance overhead.

In addition to the results of our evaluation, we argue that our approach also limits the drawbacks of common and state of the art debugging methods. We counter disorganized trial and error debugging with our systematic breadth-first search based on the scientific method. We reduce the influence of expert knowledge as we emphasize infection chains for supporting developers in creating hypotheses. If the emphasis is not sufficient, we further recommend other developers for help. Finally, instead of a symbolic debugger we offer a lightweight back-in-time debugger that allows fast access to all execution details and anomalous guidance through the large amount of trace data. Thus, we conclude that our test-driven fault navigation and incremental dynamic analysis is able to reduce time and effort required for debugging because we answer quickly where to start debugging, who understands failure causes best, what happened before failures, and which state properties are infected.

1.3. Contributions

We summarize the contributions of this dissertation as follows:

1. Test-driven fault navigation [167, 166, 197] is a debugging guide that integrates anomaly detection into a breadth-first search for systematically creating, evaluating, and refining failure cause hypotheses. With the analysis of the hidden knowledge of reproducible test cases, our approach comprises four components that support developers in following failure causes with the *scientific method*:

   Structure navigation [167] localizes suspicious system parts and so supports the *creation of (initial) hypotheses.* It emphasizes relationships between spectrum-based anomalies and provides an overview of starting points that are likely to include failure causes.

Team navigation [167] recommends other developers for helping with *creating, predicting, and later refining hypotheses.* We restrict the search space to authors of suspicious program entities only and thus we are able to suggest suitable experts even if the defect is still unknown.

Behavior navigation [167] allows developers to *experiment* with the entire execution history and to explore arbitrary object states. With the help of anomalies, it further classifies erroneous behavior for facilitating the navigation through the large amount of run-time data.

State navigation [103, 106] reveals parts of the infection chain and assists in the *observation* of and *conclusion* about failure causes. After harvesting common object properties of passing test cases, dynamically created contracts are violated by failing test cases and uncover state anomalies between failure and root cause.

2. Incremental dynamic analysis [165, 168, 103] is a novel and lightweight technique that ensures an immediate debugging experience with *test-driven fault navigation.* The following three approaches distribute dynamic analysis over multiple reproducible test runs and recording run-time data only on demand:

   Refined coverage analysis [165] allows fast access to test case coverage at methods and optional refinements at statements. Depending on developers' needs, *structure* and *team navigation* require this data on different levels of detail.

   Step-wise run-time analysis [168] splits the expensive run-time analysis of all test case execution details into an initial shallow analysis and on-demand refinements. For *behavior navigation*, developers only choose their relevant data that is automatically recharged in next test case runs.

   Inductive analysis [103] harvests and generalizes selected state properties of passing test cases such as types and value ranges. In *state navigation*, this information is able to reveal corrupted state on the infection chain of failing test cases.

3. The Path Tools framework [167, 165, 168, 197] implements our *test-driven fault navigation* and the *incremental dynamic analysis* in the Squeak/Smalltalk development environment on top of the SUnit framework. Build on top of our flexible *Path analysis framework*, the tool suite consists of our enhanced test runner *PathMap* [167, 165] for supporting *structure*, *team*, and *state navigation*, our lightweight back-in-time debugger *PathFinder* for following infection chains through *behavior* and *state* [167, 168], and our extended source code editor *PathBrowser* [197] for presenting the hidden test knowledge.

1.4. Outline

The remain

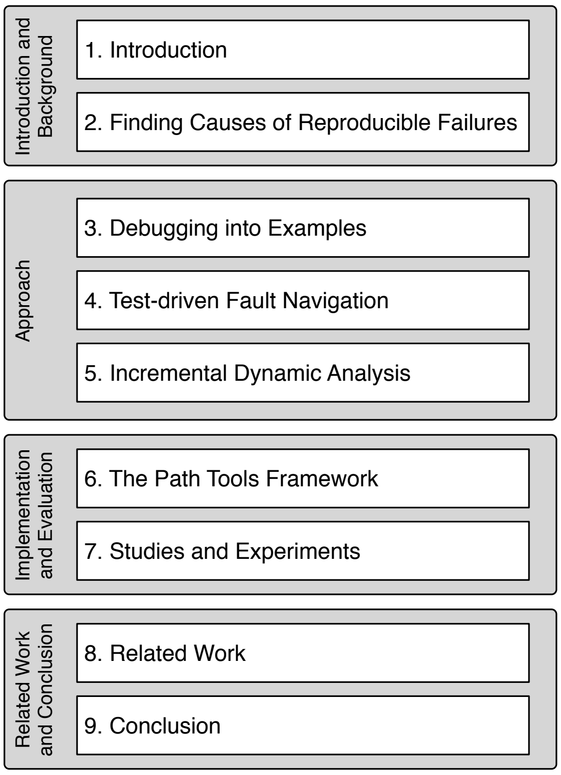

Introduction and Background
1. Introduction
2. Finding Causes of Reproducible Failures
Approach
3. Debugging into Examples
4. Test-driven Fault Navigation
5. Incremental Dynamic Analysis
Implementation and Evaluation
6. The Path Tools Framework
7. Studies and Experiments
Related Work and Conclusion
8. Related Work
9. Conclusion

Figure 1.3.: Overview of parts and chapters of the dissertation.

Chapter 2 introduces the debugging background of this work. We present a motivating failure that is used as a throughout example, define the most important debugging terms, and conclude with contemporary challenges in localizing failure causes.

Chapter 3 lays out the basic ideas behind our systematic debugging process. With the help of test cases and their hidden knowledge, we introduce our test-driven fault navigation as a breadth-first search for failure causes that assists developers with the scientific method. In conclusion, we discuss software defect types that benefit from our approach.

Chapter 4 describes the four components of test-driven fault navigation in more detail. Structure navigation identifies suspicious system parts that initially restrict the search space. Team navigation recommends other developers for help even if failure causes are still unknown. Behavior navigation allows developers to follow suspicious behavior from the observable failure back to its root cause. State navigation reveals the infection chain to support developers in creating hypotheses.

Chapter 5 presents our incremental dynamic analysis and its three different techniques that form an efficient basis for debugging with test-driven fault navigation. Refined coverage analysis collects test coverage on different levels of detail that is required for structure and team navigation. Step-wise run-time analysis enables behavior navigation to explore a complete test case execution on demand. Inductive analysis delivers common object properties of passing test cases for state navigation.

Chapter 6 outlines our Path Tools framework for the Squeak/Smalltalk development environment. We present its architecture, describe our debugging tools, and sketch the application of our incremental dynamic analysis framework. Finally, we conclude with a discussion of introducing our approach into other environments.

Chapter 7 evaluates our test-driven fault navigation and incremental dynamic analysis. We examine the practicality of our approach by conducting a user study that compares our Path Tools with common debugging techniques. To assess the effectiveness of our team and state navigation heuristics, we automatically mutate several applications with random defects and analyze the proposed results. Furthermore, we measure the efficiency of our Path Tools and the performance impact of our incremental dynamic analysis.

Chapter 8 presents related work with respect to testing, debugging, dynamic analysis, and program comprehension approaches.

Chapter 9 offers some conclusions and discusses reasonable directions for future work.

2 Finding Causes of Reproducible Failures

In this chapter, we present the essential background knowledge for our test-driven fault navigation. First, we introduce a motivating error as a continuous example that explains our approach later on (Section 2.1). Second, we define the most important terms and explain how developers systematically follow infection chains from observable failures back to their defects (Section 2.2). Finally, we summarize the challenges in testing and debugging with respect to the scientific method and conclude that there is a need to improve current approaches (Section 2.3).

2.1. Motivating Example: Typing Error in Seaside

We introduce a motivating example error taken from the Seaside Web framework [63, 169] that serves as a basis for our discussion of debugging challenges and the explanation of our test-driven fault navigation approach in the following chapters.

Seaside[1] is an open source Web framework that provides a uniform, pure object-oriented view of Web applications and combines a component-based with a continuation-based approach [177]. With this, every component has its own control flow which leads to high reusability, maintainability, and a high level of abstraction. Additionally, the fact that it is written in Smalltalk [87] allows developers to debug and update applications on the fly. It provides a layer over HTTP and HTML that allows developers to build highly interactive Web applications that come very close to the implementation of real desktop applications. Seaside consists of about 650 classes, 5,500 methods and a large test suite with more than 700 test cases (see also Section 7.1).

We have inserted a defect into Seaside's Web server and its request/response processing logic (`WABufferedResponse` class, `writeHeadersOn:` method). Figure 2.1 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 buffered response object. While the first transfers the message body as a stream, the latter buffers and sends the response as a whole. During the creation of buffered responses, there is a

[1] `www.seaside.st`

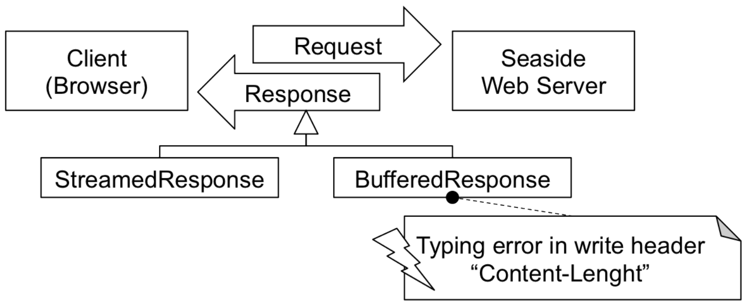


Figure 2.1.: An inconspicuous typing error in writing buffered response headers leads to faulty results of several client requests.

typing error in writing the header. The typing error 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 typing error is simple to characterize, observing it can be laborious: some clients hide the failure by tolerating corrupted header information; compilers do not report an error because the response header is built with concatenated strings; and developers tend to overlook such small typing errors in text-based source code [178].

2.2. From Failures to Defects

The way from the observable failure to its defect comprises several testing and debugging activities starting from reproducing the failure via understanding causes to correcting the defect. The *traffic* principle (track, reproduce, automate, find, focus, isolate, and correct) summarizes all these activities step by step in a systematic debugging guide [219]. As a user reports a failure, developers first *track* the problem in their bug tracker system and check if it is already known. In our Seaside example, a user complains about a specific Web browser that does not allow access to its Web application. With the help of an added problem report, developers *reproduce* the failure in their development environment. They open Seaside's start application with the mentioned browser and recognize the same failure. With this knowledge, developers implement a preferably simple test case that *automatically* reproduces the failure. For example, a test case sends a request to the start application and verifies that the response is correct. Then, developers *find* possible infection origins, *focus* on the most likely ones, and *isolate* the infection chain. In other words, they apply the scientific method and several debugging tools to follow the observable failure systematically back to its defect. In our typing error, developers have to rely primarily on Smalltalk's symbolic debugger. As this common tool supports neither back-in-time capabilities nor advice on failure causes, the creation, evaluation, and refinement of hypotheses become laborious tasks. Finally, if developers have found the root cause, they can *correct* the corresponding defect in source code and check that the failing test case now works as expected.

Regarding the traffic principle, our test-driven fault navigation deals with the hardest part of debugging, namely the localization of failure causes and defects. It builds up reproduced failures in the form of automatic test cases and guides developers along infection chains back to root causes. While our approach does not directly support the correction of defects, it does provide all necessary information to understand and solve the problem.

2.2.1. Reproducing and Testing Failures

Having a detailed problem report [33], testing is the first development activity in order to reproduce and automate the failure. Although the implementation of test cases is often not trivial, it comes with several benefits [219]. A proper test case describes the problem in an executable specification that is reproducible and automatic. It supports debugging by automatically verifying failures in less time, simplifying failure-inducing input, and documenting the system. The entire development team profits from testing because it can improve several other software development activities [6]. For the purpose of this thesis, we define testing as follows:

Testing

> "Testing is the process of determining *whether* a given set of inputs causes an unacceptable behavior in a program." [150]

Software testing is not only useful for reproducing failures, but also ensures that applications work as expected. As an essential development activity, developers specify the proper state and interaction of objects in the form of test cases. In doing so, they create large test bases that serve as safety nets for the early identification of erroneous behavior [27]. However, testing is just the start for localizing failure causes because it only verifies if an observable failure occurs or not.

2.2.2. Following the Infection Chain Backwards

To understand the problem, developers have to analyze the behavior of the failing test case completely. They identify causes and effects of the failure and find the real root cause. At the end, they can correct the corresponding source code and by this prevent similar failures. We define debugging as a two-part process:

Debugging

> "Debugging is the process of determining *why* a given set of inputs causes an unacceptable behavior in a program and *what* must be changed to cause the behavior to be acceptable." [150]

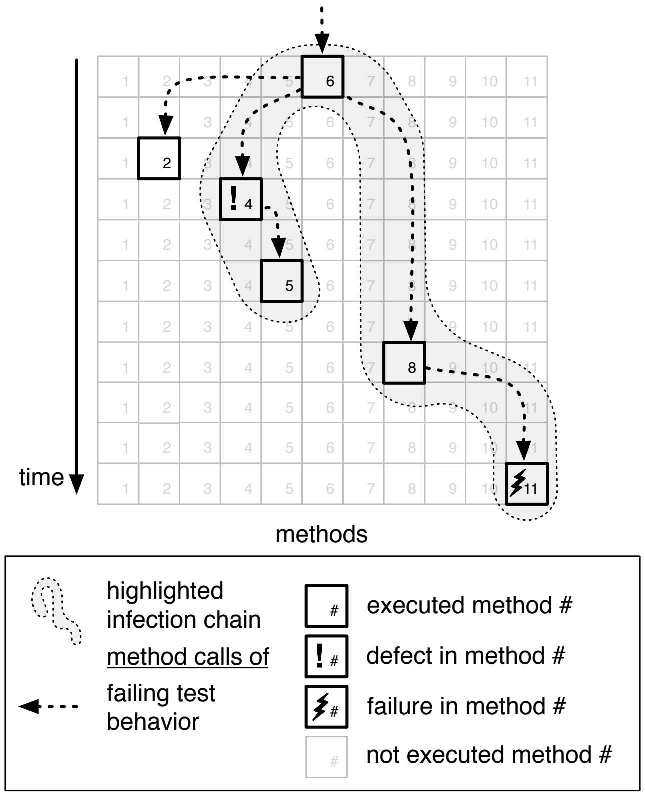


Figure 2.2.: Developers have to follow the infection chain (grey border) from the observable failure (method 11, bottom right corner) back to the defect (method 4, center left).

Although "debugging" includes the term "bug", we prevent this word because of its ambiguity. It is imprecise and can mean incorrect program code, state, or results. For that reason, we apply the following adapted definitions of failure, infection, and defect[2] [219]:

Failure

> "A failure is an externally observable incorrect program result."

Infection

> "An infection is an incorrect program state or misleading behavior."

Defect

> "A defect is an incorrect program code and corresponds to the root cause."

[2]There are similar wordings from other research communities. For example, faults correspond to defects and errors indicate infections.

All three terms together create the so-called *infection chain.* After a developer has created a defect in source code, the incorrect code is executed and causes an infection that is capable of being propagated until an observable failure is thrown. To understand how the failure comes to be, developers have to systematically follow this infection chain backwards [219]. Beginning with the failure-reproducing behavior, in the form of at least one failing test case, developers trace the observable failure via its infection chain back to the responsible defect. The small example in Figure 2.2 illustrates the infection chain with the observable failure (method 11, bottom right corner) and the defect (method 4, center left). Each row shows all eleven methods and highlights the specific method that is executed at this point in time. The grey area highlights the infection chain. Localization of the initial failure cause requires developers to decide what the corrupted state or behavior is at each executed method so that they are able to follow the infection chain backwards. These decisions are necessary to understand failure causes completely and to identify the root cause finally.

2.2.3. Correcting the Root Cause

The second part of debugging is to correct the root cause in such a way that the original or similar failures do not reappear. Although we do not focus on this topic, we argue that it is important to comprehend the entire infection chain in order to solve the problem completely [219]. For that purpose, developers isolate the infection chain and understand what causes the failure. With the help of the scientific method, they work on a diagnosis of the defect that explains previous and predicts further observations. In doing so, developers have to ensure that they find the root cause and not only an earlier failure cause on the infection chain. Otherwise, developers tend to correct rather symptoms instead of defects [219]. Thus, they can introduce new failures or the same failure occurs again under slightly different circumstances. For example, developers may solve our Seaside typing error either in the generic response class which could negatively influence also streamed responses, or they correctly fix the root cause in the write header method of buffered responses. After all, they can fix the defect, prove that the failure has gone, and make the failure-reproducing test case passed.

2.3. Challenges of Testing and Debugging

To prevent a guessing game during debugging, developers apply the scientific method to narrow down defects systematically [219]. Often debugging includes much guesswork—novice developers in particular follow more their unskillful intuition than a systematic procedure. They simply start with a depth-first search and try to debug the program here and there. This often leads to wrong decisions that require additional time and make fault localization a laborious and error-prone task. For that reason, experts systematically follow the failure back to its root cause with the help of the scientific method in the form

	Test runner and debugger	Anomalies	Expert knowledge	Back-in-time debugger	Automatic debugging
Hypothesis	-	+	o	-	+
Prediction	-	o	o	o	+
Experiment	o	-	-	+	-
Observation	o	-	-	+	-
Diagnosis	o	o	o	-	o
Section	2.3.1	2.3.2	2.3.3	2.3.4	2.3.5

Table 2.1.: Comparison of contemporary debugging approaches for applying the scientific method (+ good / o partial / - bad support).

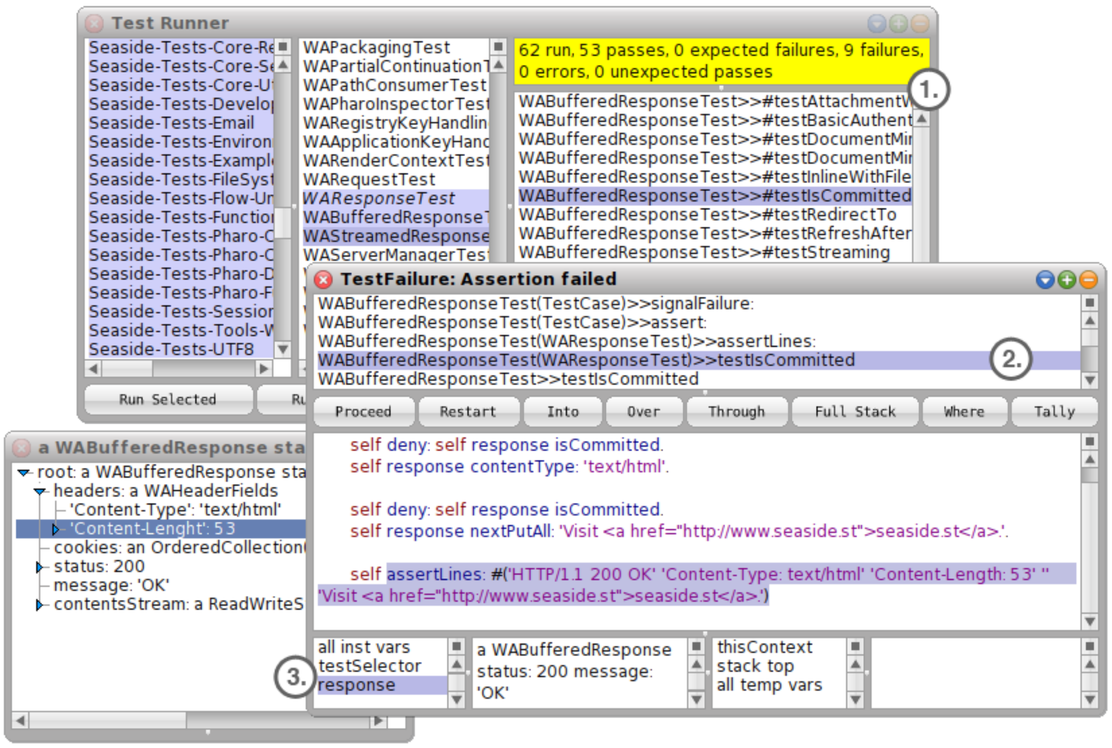
of a breadth-first search [209]. Starting with an initial hypothesis, they make predictions concerning the problem and experiment with the system to observe discrepancies in their assumptions. Based on this information, they refine or reject their hypotheses until a diagnosis is found. It has been shown that this systematic hypothesis-testing is more promising and requires less time in debugging than disorganized trial and error approaches [150, 219].

However, even though the scientific method is known as a valuable procedure for localizing failure causes, contemporary debugging approaches provide no or only partial support for it. Table 2.1 summarizes our assessment for the state of the art in debugging and its assistance during hypothesis-testing. No approach completely covers all aspects of the scientific method. Often it focuses only on one specific aspect such as creating hypotheses or experimentation. The following subsections explain each approach and its issues in general. A comprehensive discussion of related work can be found in Chapter 8.

2.3.1. Test Runner and Debugger: How to Apply the Scientific Method?

In general, debugging of test cases with standard tools faces several challenges with respect to localizing failure causes and defects. Nowadays, almost all development environments include test runners and symbolic debuggers as their debugging tools of choice. Unfortunately, these tools are not only around 50 years old [138] but they are also not well-suited for the systematic following of infection chains *backwards* to their root causes.

Test runners only execute test cases and verify if failures occur or not. There is no additional information such as differences between failing and passing tests. Hence, developers cannot restrict the search space to suspicious program entities that could help in creating initial hypotheses and making predictions. Furthermore, they can only experiment with test runners to a limited extent. For example, developers are not able to observe which parts of the program are being executed. This task typically requires other



```
Seaside-Tests-Core-Re
Seaside-Tests-Core-Se
Seaside-Tests-Core-U
Seaside-Tests-Develo
Seaside-Tests-Email
Seaside-Tests-Environ
Seaside-Tests-Exampl
Seaside-Tests-FileSyst
Seaside-Tests-Flow-Ur
Seaside-Tests-Functio
Seaside-Tests-Pharo-C
Seaside-Tests-Pharo-C
Seaside-Tests-Pharo-D
Seaside-Tests-Pharo-F
Seaside-Tests-Sessior
Seaside-Tests-Tools-W
Seaside-Tests-UTF8
```

```
WAPackagingTest
WAPartialContinuationT
WAPathConsumerTest
WAPharoInspectorTest
WARegistryKeyHandlin
WAApplicationKeyHand
WARenderContextTest
WARequestTest
WAResponseTest
WABufferedResponseT
WAStreamedResponse
WAServerManagerTest
```

```
62 run, 53 passes, 0 expected failures, 9 failures,
0 errors, 0 unexpected passes
WABufferedResponseTest>>#testAttachmentW
WABufferedResponseTest>>#testBasicAuthent
WABufferedResponseTest>>#testDocumentMi
WABufferedResponseTest>>#testDocumentMi
WABufferedResponseTest>>#testInlineWithFile
WABufferedResponseTest>>#testIsCommitted
WABufferedResponseTest>>#testRedirectTo
WABufferedResponseTest>>#testRefreshAfter
WABufferedResponseTest>>#testStreaming
```

```
WABufferedResponseTest(TestCase)>>signalFailure:
WABufferedResponseTest(TestCase)>>assert:
WABufferedResponseTest(WAResponseTest)>>assertLines:
WABufferedResponseTest(WAResponseTest)>>testIsCommitted
WABufferedResponseTest>>testIsCommitted
```

```
    self deny: self response isCommitted.
    self response contentType: 'text/html'.

    self deny: self response isCommitted.
    self response nextPutAll: 'Visit <a href="http://www.seaside.st">seaside.st</a>.'.

    self assertLines: #('HTTP/1.1 200 OK' 'Content-Type: text/html' 'Content-Length: 53' ''
'Visit <a href="http://www.seaside.st">seaside.st</a>.')
```

```
root: a WABufferedResponse sta
  headers: a WAHeaderFields
    'Content-Type': 'text/html'
    'Content-Lenght': 53
  cookies: an OrderedCollection
  status: 200
  message: 'OK'
  contentsStream: a ReadWriteS
```

```
all inst vars
testSelector
response
```

```
a WABufferedResponse
status: 200 message:
'OK'
```

```
thisContext
stack top
all temp vars
```

Figure 2.3.: Localizing failure causes with standard tools (Unit test runner (1), symbolic debugger (2), and object explorer (3)) is cumbersome.

tools such as symbolic debuggers. Only the test result feedback is helpful for a diagnosis because the correctness of fixes is directly reflected in passing test cases.

Symbolic debuggers suffer from missing advice on failure causes and back-in-time capabilities. They only allow developers to stop a program and to access the run-time stack at a particular point in time. Neither do they report what is going wrong, nor do they offer capabilities to follow infection chains backwards. A probable outcome of this is that developers rely primarily on their intuition for creating hypotheses and experiment with the system only in the forward direction even though the defect is located in the past. Since it is hard to understand failure causes and how they come to be, we argue that developers rather follow a disorganized trial and error when debugging with these tools.

Also, the localization of our typing error with standard tools is cumbersome. Figure 2.3 depicts a typical debugging session after the observable failure has been reproduced by several test cases. Seaside's test suite answers with 9 failed and 53 passed test cases for all response tests (1). Since all failing runs are part of `WABufferedResponseTest`, developers might expect the cause within buffered responses. However, this hypothesis lacks evidence such as a list of covered response methods being executed by all failed tests. Experimenting with the standard debugger on a failing test shows a violated assertion

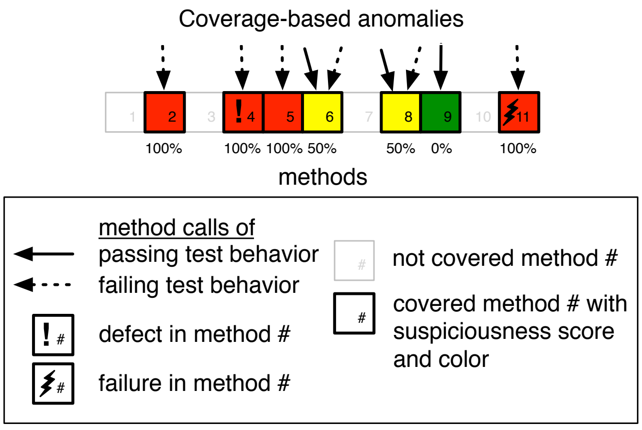


Figure 2.4.: Spectrum-based fault localization reveals a set of suspicious source code entities that could be responsible for failure causes.

within the test method itself (2). This, however, means that developers only observe the failure instead of its origin. Only the current stack is available, but our typing error is far away from the observable malfunction. The thrown assertion suggests that something is different from the expected response (3). Developers have to introspect the complete response object for localizing the typing error. There are no pointers to the corrupted state or its infection chain. Remarkably, the response status is still valid (`200, OK`). 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 Seaside's continuation-based communication [177].

2.3.2. Anomalies: Where to Start Debugging?

Anomalies support the creation of initial hypotheses by comparing program spectra. Program spectra represent which program parts were active during a specific execution and include several behavioral information from method coverage to state predicates [101]. *By comparing spectra of passing and failing test cases, differences reveal anomalies as a property that differs from expectations and is likely to include failure causes.* Not each anomaly is a failure cause, but its occurrence can be a valuable indication. For example, spectrum-based fault localization techniques [2, 122] reveal anomalies by comparing the differences of test case coverage. These approaches produce a prioritized list of suspicious statements which restricts the search space and reduces initial speculations for creating proper hypotheses.

Figure 2.4 shows the analysis of the spectrum-based anomalies for our previous infection chain example. The static structure shows the eleven methods (represented as boxes) in their source code definition order. We compute a percentage value determining the failure cause probability for each method with the help of the Tarantula metric [122]. In other words, the higher the number of covered failing with respect to passing test cases the higher its probability to include failure causes. For example, method 2 is only covered by one failing test and owns the highest suspiciousness score, while method 6 is covered by both and has a 50 % failure cause probability. These suspiciousness scores can also be mapped to colors. So, each covered method box is highlighted with its failure cause probability from red (high) to green (low). Finally, the set of suspicious source code entities restricts the search space to 4 out of 11 methods and so supports developers in creating initial hypotheses.

Although anomalies restrict the search space by providing excellent hints for finding hypotheses, they provide no further information for evaluating them. Unfortunately, anomalies are not failure causes—developers only have numerous starting points that must be debugged one by one. They offer only an initial indication that is able to limit the search space to around 20 % of the program [2]. By experimenting and observing these suspicious source code entities with standard tools, anomalies and failure causes are not related to each other. Thus, developers have to deal with a number of difficult questions to make predictions and to follow infection chains backwards: which anomalies include failure causes or maybe the defect; what are the relations between anomalies and failing test cases; how are infected state and anomalous behavior propagated so that failures come to be. In particular, the delocalization issue of object-oriented programming languages makes a diagnosis non-trivial because the required information is spread all over the source code [65]. In our small example in Figure 2.4, there are four methods with the same high suspiciousness value (red color) and it is not clear how they are related to failure causes, to each other, and which statements are responsible for the failing behavior. For example, it is not clear that method 2 is executed before the defect (method 4) and thus works as expected (compare to Figure 2.2). In addition to this problem, spectrum-based fault localization techniques also suffer from a scalability problem [121]. Either they provide anomalies at the statement-level, whose dynamic analysis comes along with a perceivable performance decrease [213] and creates numerous results, or they include other program entities, such as methods or classes, which miss important fine-granular information.

Our Seaside typing error reports 54 anomalous methods including four anomalies with the highest suspiciousness score. All four methods are part of the buffered response class and so focus hypothesis-testing to this source code snippet. However, developers do not know which method includes the root cause; how anomalies are related to each other; and how the infection chain looks like. Existing techniques do not consider the infection chain at all and so developers have to laboriously debug unrelated anomalies one by one. As defects do not often come first, analyzing spectrum-based anomalies can be time-consuming and strongly depends on program comprehension.

2.3.3. Expert Knowledge: Who Understands Failure Causes Best?

For the debugging of failure causes or the interpretation of suspicious entities, developers' expertise significantly influences the required effort [11]. More experienced developers create better hypotheses and predictions, require fewer iterations, and reveal failure causes faster than novices that do not know the code base [93, 210]. There are several automatic approaches that deal with the identification of proper expert knowledge with respect to a reported failure. They analyze bug and source code repositories and assign new failures to similar bug reports and their previous developers.

Unfortunately, the identification of corresponding experts is quite challenging since observable failures do not explicitly reveal infected system parts. Although we seek out experts for understanding the infection chain with its failure causes and the defect, existing approaches only analyze observable failures. For that reason, recommended developers are seldom good matches for debugging specific failures. Furthermore, these approaches consider either the entire source code or similar failure reports to automatically assign bug reports to more experienced developers. Thus, they consider a too large search space or they require a comprehensive bug tracker database with already fixed failures. Even if proper experts are found, these developers still have to rely on cumbersome debugging tools for observing and experimenting with a program's run-time.

In our Seaside example, we are looking for an expert that understands the processing of buffered responses. However, with a bug report about invalid results in a specific browser, we will probably find developers with a strong background in Web browsers. Even though this result is not bad, we argue that the original developers of buffered responses would localize failure causes more easily because they better know what is expected and unexpected behavior.

2.3.4. Back-in-time: What Happened before Failures?

In contrast to symbolic debuggers, back-in-time debuggers help to understand complete infection chains and to follow failure causes back to their defects. As the reasons for observable failures happened in the past, these debuggers record all run-time information before the failure occurs and then present it afterwards. Developers start with the observable failure, step backward, and search for infections at each point in the program's execution history. By asking questions about a program's run-time such as: "Where did the value of a variable change?", they can develop a deeper understanding of infection chains step by step until it is clear how failures come to be. Since back-in-time debugger provide access to the complete infection chain, they are particularly suitable for the experimentation and observation activities of the scientific method.

However, starting debugging at observable failures still compels developers to analyze a great number of failure causes and their effects until they found the root cause [137].

Back-in-time debuggers do not support a direct navigation along the infection chain [104]; developers have to examine an enormous amount of data manually in order to create proper hypotheses. The missing classification of suspicious and harmless behavior leads to numerous and often laborious decisions which execution subtree to follow [219]. Furthermore, most back-in-time debuggers often come with a performance overhead [135] or a more complicated setup [175] that does not allow a seamless and immediate access to run-time information [201]. Unfortunately, this overhead renders these tools rather impractical for frequent use. Thus, the infection chain is hard to follow, developers require much time, and debugging the entire test case execution becomes a laborious activity.

In our small Seaside example, a back-in-time debugger helps in revealing the execution history with the drawbacks of missing guidance along the infection chain and a dramatically slow-down during test case execution. Post-mortem back-in-time debuggers for Smalltalk [108, 142] produce an enormous amount of data because they record large parts of the run-time behavior and state changes until the test case fails. Although this information contains everything that is required for localizing the infection, developers have no support for systematically browsing the large amount of data and to create proper hypotheses. Moreover, the recording of all run-time events can slow down the program execution by a factor of up to 300 [135]. The most simplified test case requires around 200 ms for its pure execution and around one minute with a back-in-time debugger. For these reasons, we argue that guidance and immediate access to run-time behavior are important in order to prevent time-consuming back-in-time debugging sessions.

2.3.5. Automatic Debugging: Which State Properties Are Infected?

Automatic debugging approaches support developers in identifying failure causes or even the defect. There are several approaches that reveal parts of the infection chain automatically. Likely invariants [70] derive generalized contracts from passing test cases and compare them with failing test cases. Differences reveal state anomalies with a high probability of including failure causes. Delta debugging [217] pinpoints causes between similar failing and passing test cases until the defect is found. Algorithmic debugging [193] partially automates the debugging process by systematically creating hypotheses. All of these approaches help in narrowing down the search space by answering which state properties are infected. They provide valuable hints for creating hypotheses and support the prediction by revealing parts of the infection chain.

In general, automatic debugging approaches are able to find failure causes, but the complete identification of defects often requires more investigation. Although they are able to shorten the infection chain, developers still have to experiment and observe the rest of a failing behavior to finally find a proper diagnosis. During these steps, it is hard to understand the entire infection chain because its first part is often unknown and the remaining part still has to be analyzed with inconvenient debugging tools. What is more, all automatic approaches have some drawbacks with respect to their quality and scalability. The analysis of likely invariants requires both a comprehensive test base and a vast amount of time. Delta debugging expects a high similarity between failing and passing test cases, otherwise its results are imprecise. Finally, algorithmic debugging has not been shown to be efficient in real-world applications because it does not scale with complex object structures [219].

In our Seaside example, the automatic approaches identify some infected state properties but they cannot localize the defect. Likely invariants produce a set of unrelated state anomalies that may help in starting debugging. Delta debugging reduces the response object to the typing error but without revealing the corresponding method. Declarative debugging creates an initial hypothesis about the wrong response objects but misses valuable details about the typing error.

2.4. Summary

We introduced the background in testing and debugging being specific for our work. We started with a motivating example describing a typing error in the request/response processing of the Seaside Web framework. After that we defined the most important terms and ideally followed the infection chain from the observable failure to its defect. Finally, we discussed the challenges with the state of the art in debugging and that these approaches provide no or only partial support for the scientific method.

We argue that hypothesis-testing is a promising method for systematically localizing failure causes, but there is a need for improving current debugging approaches. The search for defects requires the integration of several perspectives in order to comprehensively support the scientific method. Existing approaches are limited to a particular debugging task so that the further investigation requires manual work. For example, back-in-time debuggers provide access to the entire execution history but they do not reveal the infection chain in the large amount of run-time data. For these reasons, this evidence is conclusive that developers need a more comprehensive and integrated support for hypothesis-testing. Hence, they are able to answer important debugging questions on their way to root causes, such as how to apply the scientific method; where to start debugging; who understands failure causes best; what happened before failures; and which state properties are infected?

Part II.

Approach

3 Debugging into Examples

In this chapter, we present the basic ideas behind our test-driven fault navigation for improving debugging of failure-reproducing test cases. We start with the essential assumption that test cases yield a hidden and valuable source of information that is able to enhance debugging of reproducible failures (Section 3.1). Based on this idea, we introduce the systematic top-down debugging process of our test-driven fault navigation which integrates the hidden test knowledge in order to guide developers to failure causes (Section 3.2). Finally, we discuss the types of software defects which our novel test-driven fault navigation approach is applicable to (Section 3.3).

3.1. Leveraging Test Cases for Debugging

Although test cases have several benefits, developers often pay little attention to them for debugging and localizing defects. Every test case can be understood as an executable specification of a requirement and its expected program behavior. Developers can execute them as often as necessary and check if failures occur. For this reason, test cases have to be reproducible, fast, and without any side-effects after their execution. However, developers only consider the results of test cases that reveal observable failures. Even though failing tests include all information required for analyzing entire infection chains [182], developers utilize them only to a limited extent for localizing failure causes. The first part of Figure 3.1 illustrates this current practice. On the left, several test cases exercise parts of the system under observation on the right. In doing so, some test cases report failures and errors and reveal their last point of execution, namely the observable failure. This point relates to a violated assertion or exception and is often far away from the failure-inducing root cause. As there are no obvious relations between defect and failure, developers cannot follow infection chains of failing test cases backwards. They only have symbolic debuggers that start the debugging session at the failure without any possibilities to get back to the defect.

We propose two new perspectives that also consider test cases as behavioral *paths* through the system. First, we see test cases as entry points into reproducible behavioral examples [197]. By running a specific test case and analyzing its execution path, developers make sense of the entire behavior and debug into examples. The second part of Figure 3.1 shows the uncovered behavioral paths of our test cases. As all failing tests include the

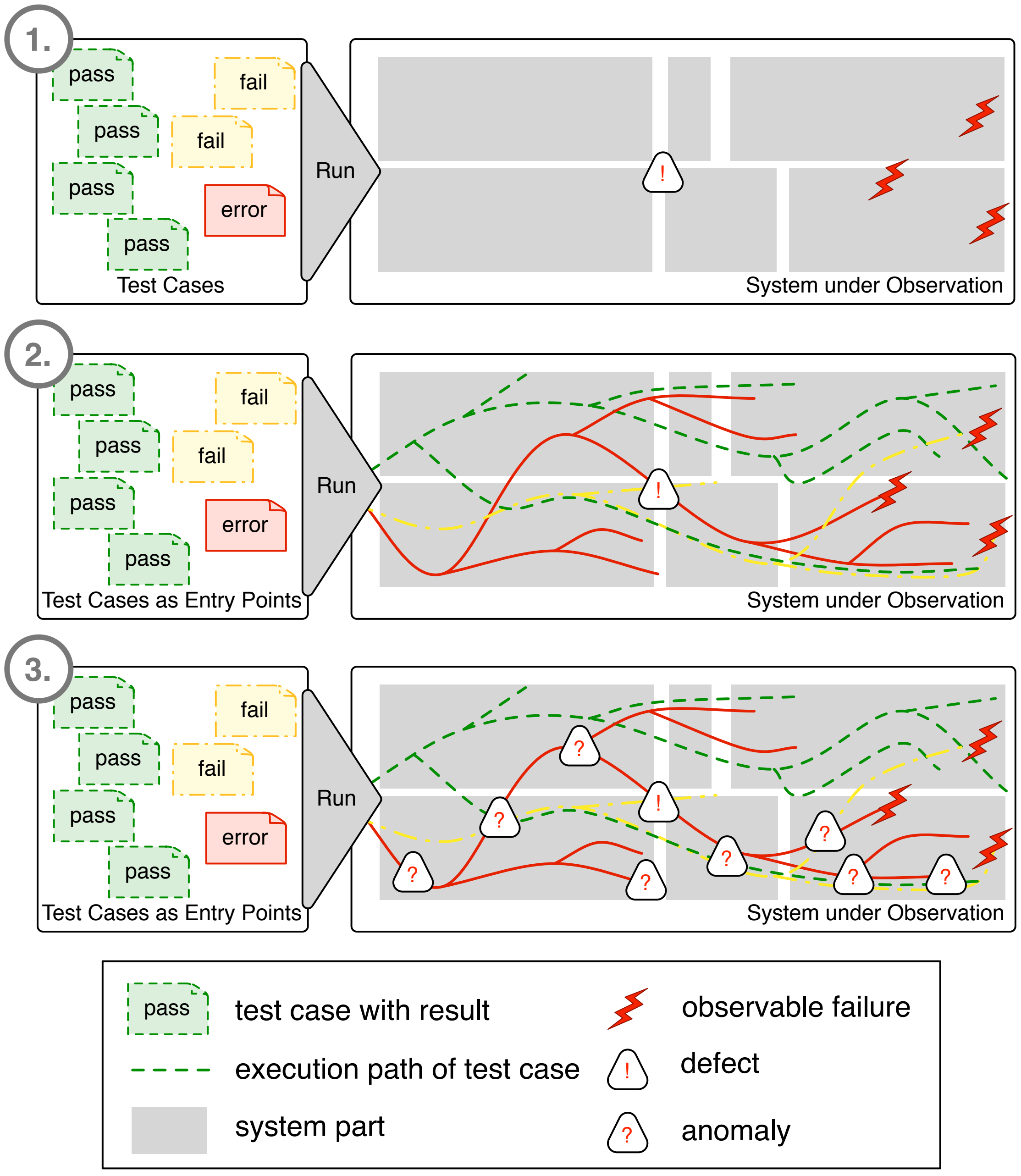


Figure 3.1.: So far, test cases only report their results (1). With our new perspectives, we also consider test cases as reproducible entry points into behavioral examples (2) and reveal their hidden knowledge (3).

defect, we allow developers to follow specific infection chains back to their root causes. Second, passing test cases verify not only the correctness of their explicit assertions, but also define implicit assertions along their execution [103]. If a test case is valid, then all executed program statements and used objects can also be seen as sufficiently correct. Comparing this hidden test knowledge with failing test paths automatically reveals anomalies and parts of the infection chain. In the third part of Figure 3.1, uncovered anomalies guide developers along executed test cases. Both of these new perspectives give a valuable source of information for debugging failing test cases back to their defects.

3.1.1. Entry Points into Reproducible Behavioral Examples

We suggest test cases as entry points into behavioral examples that support developers in comprehending and debugging programs. The IEEE glossary defines an entry point as "a point in a software module at which execution of the module can begin" [159]. Following this definition, we regard test cases as natural entry points into source code because they are executable specifications of expected system behavior that is further reproducible and deterministic [66, 149].

To provide developers access to these entry points not only at the test case definition but also at arbitrary methods of the system under observation, we need to uncover the implicit test coverage relationship. In order to establish these links, we trace and record all methods that are executed during a test case run. Having these run-time data, we can provide the set of test cases that cover a specific method. Thus, we make coverage information explicit and provide developers with the required entry points to trigger an execution path and debug into source code at arbitrary program entities.

With entry points all over the program, developers can better understand the abstract source code with small and concrete examples. Usually, comprehending source code is an essential and important part of software development. Extending, modifying, and debugging a system requires an in-depth understanding, ranging from the intended use of interfaces to the interplay of multiple, interdependent objects. However, program comprehension from source code alone is difficult because it abstracts from concrete execution paths, every single class and method contributes to a large-scale collaboration of run-time objects, and object-oriented language concepts such as late binding make following a message flow more difficult [65]. As behavioral properties can only be determined precisely at run-time [14], we argue that developers require execution examples to understand programs entirely. Such examples can be provided by test case entry points that allow developers to inspect how a covered method is used in reality. If developers are interested in a specific example, we start a covering test case, interrupt its execution at the corresponding method, and offer the opportunity to comprehend source code by means of debugging tools. Developers can now understand method arguments, collaborating objects, and the run-time stack [197].

Apart from behavioral examples at one specific method, test cases also provide access to complete execution paths. A test case executes one specific path through the system which can be recorded with all its behavior and state to understand the complete execution history. In other words, during debugging a failing test case, its path also contains the important infection chain with the observable failure and the defect. However, traditional approaches for analyzing a program's run-time are time-consuming and expensive. They capture comprehensive information about the entire execution up-front, which is in large parts not required at all. We solve this problem with the reproducible and deterministic properties of test cases. We assume that a test case always takes the same path through the system so that each execution comprises the same behavior and state information. With this insight, we are able to split the expensive analysis of a program's run-time over multiple test runs. Based on developers' decisions, we divide the analysis into multiple steps: A high-level analysis followed by on-demand refinements. Thus, we incrementally collect only the data developers are interested in and so reduce the analysis overhead to a minimum while preserving instantaneous access to detailed information.

3.1.2. The Hidden Test Knowledge

Each test case path includes numerous method calls and state changes which are a valuable but hidden source of information for several software engineering activities. During their execution, test cases check not only the explicit assertions in their definitions but also establish implicit assertions in all other used program entities. As long as test cases finish with correct results, we assume that everything on their execution paths is also valid[1]. Thus, we can inductively derive from the specific and correct run-time values more generalized properties that reveal these implicit assertions. With this hidden test knowledge, developers have additional information for understanding their programs. For example, our type harvesting exploits test cases to obtain type information for a code base automatically [103]. We derive type data from the concrete objects used during test case executions and provide developers this hidden information to assist in navigating source code and using application programming interfaces (APIs) correctly.

The hidden test knowledge can also support debugging by comparing its generalized and valid state properties with failing test cases. We convert the revealed but still implicit assertions into explicit contracts for all program entities of passing test cases. If contracts are violated during the execution of failing test cases, then they reveal anomalies that can highlight parts of the infection chain. Such anomalies describe differences between execution paths in their behavior and state properties that have a high probability to include failure causes [70, 121]. Thus, they provide developers helpful advice to reduce their speculations and strengthen their hypotheses.

[1]Even if defects can cancel and so do not result in observable failures, we argue that this assumption is sufficiently correct in the vast majority of cases [70, 121]

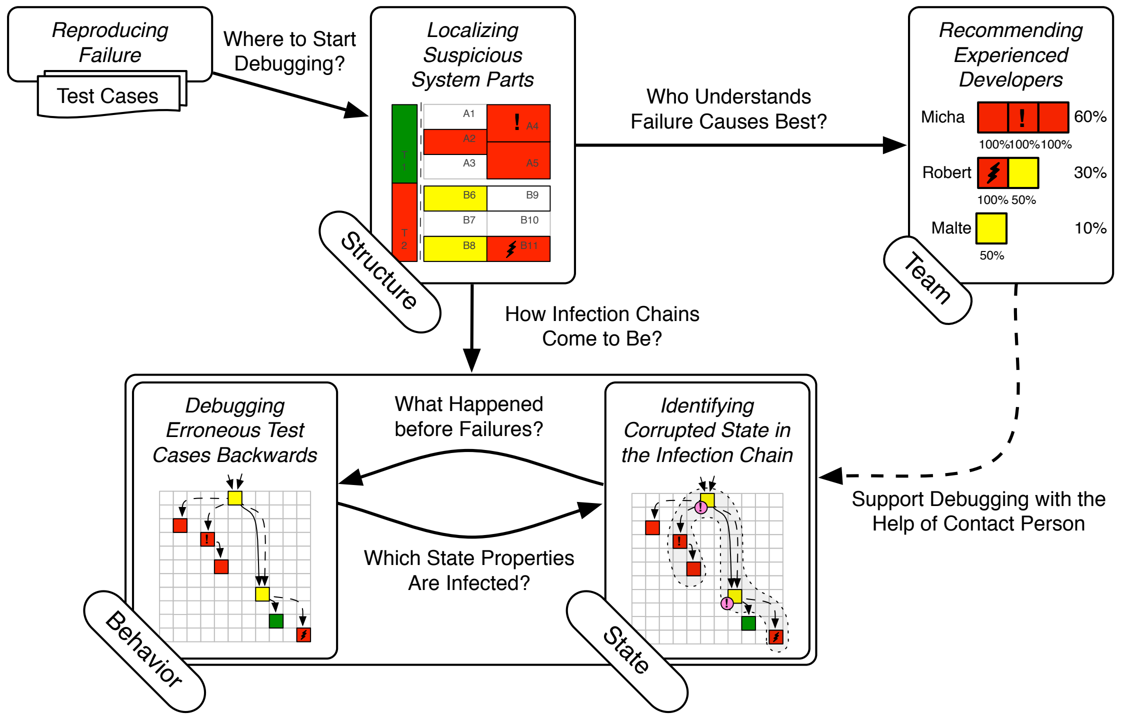

Reproducing Failure
Test Cases
Where to Start Debugging?
Localizing Suspicious System Parts
Structure
Who Understands Failure Causes Best?
Recommending Experienced Developers
Micha 60%
100% 100% 100%
Robert 30%
100% 50%
Malte 10%
50%
Team
How Infection Chains Come to Be?
Debugging Erroneous Test Cases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

Figure 3.2.: 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.

3.2. A Debugging Guide based on Anomalies

With the help of the new perspectives on test cases and our experiences from the challenges in testing and debugging, 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 [209] but also integrates the 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 systematic hypothesis-testing. Figure 3.2 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. Apart from 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 [27]. In the case of our Seaside example, developers have to implement a simple integration test that sends a server request and waits for a corrupted response which cannot be parsed correctly.

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 spectrum-based anomalies at methods and refines results at statements on demand. In our Seaside example, all failing tests overlap within response handling classes and failure causes of our typing error can be isolated within a few methods.

Recommending experienced developers (*Team navigation*) Some failures require additional expertise 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. In our example, the developer ranking metric proposes contact persons who have recently worked on the most suspicious buffered response methods.

Debugging erroneous test cases backwards (*Behavior navigation*) To refine the 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. In our example, developers examine the request-response processing of a failing test in detail. Due to a classified trace, they can shorten the search for corrupted behavior to the creation of buffered response objects.

Identifying corrupted state in the infection chain (*State navigation*) In addition to the classification of executed behavior with spectrum-based anomalies, we 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 likely 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. In our Seaside typing error, we first collect likely invariants from all streamed response tests and then execute our failing test case. Thereby, we obtain two arguments with a spell checker violation close by the defective method.

Apart from the systematic top down process for debugging reproducible failures, the combination of test cases 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 helps developers to answer 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 high cost 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 state properties are infected.

3.3. Limitations of Software Defect Types

In general, our test-driven fault navigation approach is applicable to software defects that can be reproduced by at least one test case. As developers follow this failing test case back to its defect step by step, this test case also needs to be deterministic for our approach. In other words, it always executes the same behavior and creates the same state so we can ensure that our incremental dynamic analysis works correctly and provides the requested results on demand. In most cases, this test case property is valid because most parts of the system work deterministically. If this is not the case, techniques such as record and replay [48] are able to make non-determinism reproducible by storing the first return value and answering the same value in all further runs. In addition to the reproducibility of test cases, our approach further requires a large test base. Without enough passing test cases, the reference test knowledge for our structure and state navigation is missing and we cannot reveal anomalies that guide developers along the infection chain. Nevertheless, developers are still able to debug the entire execution history of a failing test case.

There is a comprehensive classification scheme for software defects that we can also satisfy with our approach [150]. This scheme describes for all kinds of programming languages the most important reasons for failures and distinguishes between design and coding errors. While the first describes defects during the design phase such as wrong data structures, algorithms, and specifications, the latter considers defects during the coding phase such as problems with object-orientation, dynamic data structures, and control flow. The root cause checklist consists of five design and 15 coding error categories among which we can completely localize four design and 13 coding error categories without any problems. The remaining design error category deals with hardware problems which we cannot reproduce and check easily. The coding error categories (finalization errors and memory problems) are not relevant in our context because Smalltalk includes automatic memory management. Except the hardware problems, we argue that the other two software defect categories can also be localized with our test-driven fault navigation because they also satisfy the requirement of being reproducible.

However, there are also some defects that our approach cannot entirely handle so far. First, we are limited to one thread of control and thus we cannot correct concurrency issues with multiple processes or network communications. Second, not every non-determinism behavior can be reproduced by record and replay approaches so that our tools do not work reliably. Finally, Heisenbugs [219] are not observable by debugging tools because their analysis changes the corresponding failing behavior. During the development and evaluation of our approach, we have not found any of these specific defects. Nevertheless, we consider such problems as a valuable direction for future work.

3.4. Summary

We answered the question what we can learn from testing in order to improve debugging. We introduced two new perspectives for test cases and their included execution paths. First, we considered them as reproducible and deterministic entry points into behavioral examples that grant incremental access to entire execution histories. Second, we revealed the hidden test knowledge by deriving generalized program properties from valid execution paths which uncover anomalies in failing test cases. Based on these new perspectives, we presented our test-driven fault navigation as a debugging guide to localize failure causes in failing test cases. Our systematic debugging process combined test cases, their execution histories, and the hidden test knowledge into a systematic breadth-first search. With the help of anomalies, four navigation techniques guided developers to failure causes in structure, team, behavior, and state of software systems. Finally, we discussed the limitations of our perspectives, approach, and its assumptions with respect to various software defect types. In doing so, we found out that we are able to debug most reproducible failures with our test-driven fault navigation.

4 Test-driven Fault Navigation

In this chapter, we introduce each of our four test-driven fault navigation techniques in more detail. Structure navigation localizes suspicious system parts and restricts the search space for creating initial hypotheses (Section 4.1). Team navigation recommends experienced developers that can help with debugging a specific failure even if the root cause is still unknown (Section 4.2). Behavior navigation guides developers along infection chains by debugging failing test cases back in time and highlighting erroneous program behavior (Section 4.3). State navigation identifies corrupted object properties and automatically reveals parts of infection chains (Section 4.4).

4.1. Structure Navigation: Localizing Suspicious System Parts

To support the creation of initial hypotheses, our structure navigation provides an overview of the system under observation that highlights problematic areas and restricts the search space. Developers are able to perceive at a glance relations between all methods and their probability to include failure causes. This enables feedback about suspicious system parts so developers know where to start debugging.

With the help of spectrum-based fault localization [2, 122], which predicts failure causes by the ratio of failed and passed tests at covered program entities, we analyze overlapping test behavior, localize suspicious methods, and visualize their results in form of a compact and interactive tree map. In doing so, our structure navigation allows not only fast access to spectrum-based anomalies within methods but also on-demand refinements to more expensive results concerning statements. As the method-level and our optional refinements provide a good trade-off between performance and comprehensibility, we offer a scalable solution for spectrum-based fault localization in larger systems.

4.1.1. Compact System Overview of Classes and Methods

A structural system overview and its relation to test execution in the form of a compact and scalable tree map [119] allows for a higher information density compared to a list or class diagram. Figure 4.1 presents a schematic tree map visualization for the system structure of our infection chain example. The visualization represents packages as columns

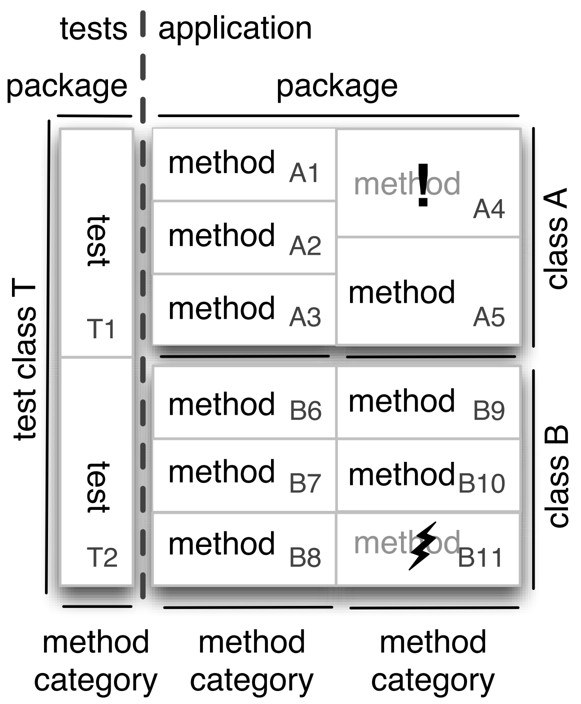


Figure 4.1.: A schematic tree map visualization for a system structure consisting of packages, classes, method categories, methods, and tests.

and their classes as rows. Each class represents each of its methods as a box. In addition to it, methods are ordered in columns with respect to their method category[1]. The allocated space is proportional to the number of methods per node. Packages, classes, and methods are sorted alphabetically and for a clear separation we distinguish between test classes on the left-hand side and application classes on the right-hand side. The alphabetic organization makes finding a particular element simple, even for large systems. Developers can systematically explore the visualization and interactively obtain more details about specific method boxes such as its name and covering tests. Moreover, each method can be colored with a hue element between green and red for reflecting its suspiciousness score and a saturation element for its confidence. As a result, a method box with a high failure cause probability possesses a full red color and requires attention from developers. In addition to that, the visualization colors a test method in green, yellow, or red if the test respectively succeeds or fails.

The tree map visualization can represent applications with thousands of methods on a standard screen because of a high information density at minimal required display space. For example, a tree map that allows minimal boxes of 4×4 pixels is able to scale up to 4,000 methods on 500×500 pixels space. Even though this should suffice for most medium-sized applications, we allow for filtering specific methods such as accessors and

[1]We provide Smalltalk's method categories as an optional layer, too.

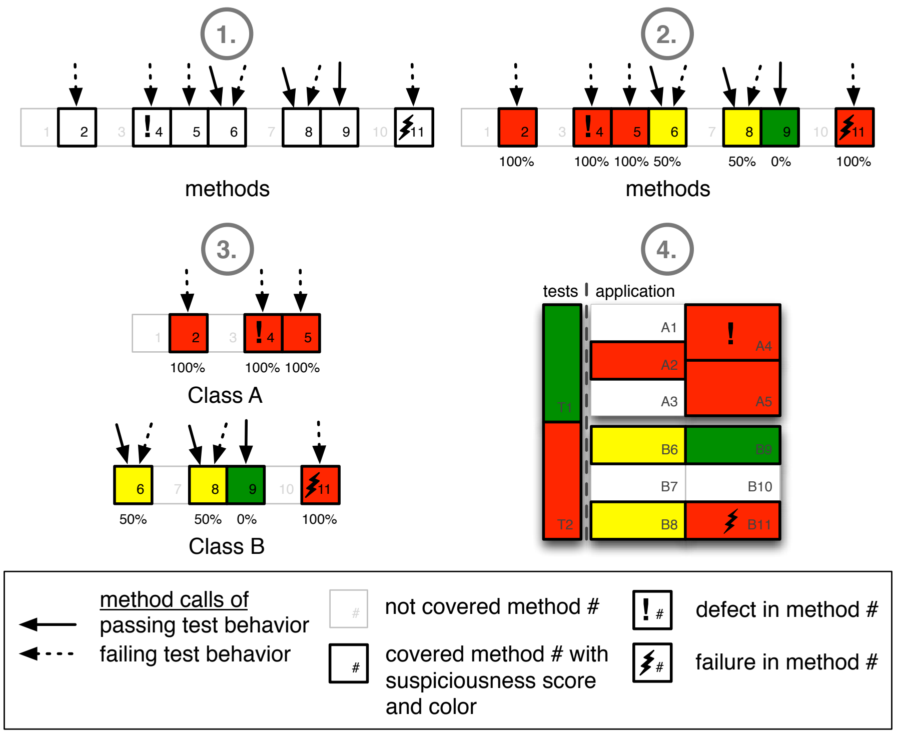


Figure 4.2.: Based on covering tests per method (1), spectrum-based fault localization reveals anomalies by computing suspiciousness scores for each program entity (2). Our structure navigation maps these results onto the system structure (3) and summarizes suspicious system parts in the form of a tree map (4).

summarizing large classes to cope with even larger systems. If methods have not enough space in a class, a developer can open the corresponding box to obtain a new and separate tree map visualization dedicated to the class and all its implementation details.

4.1.2. Spectrum-based Anomalies of Unit Tests

Our structure navigation originates in the spectrum-based fault localization approach [2, 122] and maps their revealed anomalies onto our compact system overview. Figure 4.2 summarizes our structure navigation including the comparison of passing and failing tests at covered methods, the computation of spectrum-based anomalies, and the localization of suspicious system parts.

We require for each method the number and results of their covering test cases. For that reason, we run all test cases, collect which methods are being executed, and store the final test result. So, we create a mapping between a test case, its result, and all involved methods. For example, method 6 has been executed by a passing and a failing test case. To reduce the performance overhead of run-time observation, our dynamic analysis restricts the instrumentation to relevant system parts and 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. Such partial traces [91] are also the foundation for rendering our structural tree map.

With the help of the collected test coverage, spectrum-based fault localization [2, 122] automatically identifies anomalies and colors suspicious program entities in source code. Spectrum-based anomalies and their failure cause probabilities are estimated by the ratio of all failing tests to the number of 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. Each method distinguishes between a *suspiciousness* and a *confidence* score. While the former scores the failure cause probability with respect to covered tests and their results, the latter measures the degree of significance related to the number of all test cases. With time, several metrics for spectrum-based fault localization have been proposed among which *Ochiai* has shown to be the most efficient one [2].

$$\mathit{suspiciousnessOf}(m) = \frac{\mathit{failedAt}(m)}{\sqrt{\mathit{totalFailedTests} * (\mathit{failedAt}(m) + \mathit{passedAt}(m))}}$$

This formula divides the number of failing tests at a method m by the square root of the number of all failing tests multiplied with the sum of the number of failed and passed tests at the same method. If at least one failing test is available, it returns a value between 0 and 1 for each tested method. To visualize these results, spectrum-based fault localization further colors covered program entities with a hue value between green (120) and red (0). For example, method 6 has a suspiciousness score of 0.5 and a yellow color.

$$\mathit{hueOf}(m) = 120 * (1.0 - \mathit{suspiciousnessOf}(m))$$

To assess the significance of a suspiciousness value, we apply a slightly adapted confidence metric. It only considers the relation between failed tests per method and all failing tests in order to hide correct behavior for fault localization.

$$\mathit{confidenceOf}(m) = \frac{\mathit{failedAt}(m)}{\mathit{totalFailedTests}}$$

The returned value is directly mapped to the saturation component of already colored method nodes. By looking only at faulty entities, we reduce the visual clutter of too many colors and results. For example, a method covered by three out of six failing tests is grayed out. Finally, the entire source code can be colored with anomalies and developers see at first sight which program entities are potential failure causes. Unfortunately, the source code organization does not primarily reflect the system architecture. Hence, the understanding of unrelated and wide-spread anomalies tends to be difficult and requires a vast amount of time.

For these reasons, we combine spectrum-based anomalies with a compact system overview to further reveal relationships between anomalies and the architecture. We analyze the source code and identify program entities that belong together. Packages, classes, method categories, and methods help to group anomalies and to show their interconnections to the logical program structure. For example in Figure 4.2, there are three very suspicious anomalies in class A, while class B is also checked by passing test cases. To further increase the information density, a compact system overview summarizes all spectrum-based anomalies in the form of a tree map. This visualization allows for scalability with respect to the system architecture, anomalies, and their relationships. So, developers obtain a better overview about suspicious system parts which lowers the effort for program comprehension and restricts the search space for initial hypotheses.

Adapting spectrum-based fault localization to unit testing limits the influence of multiple faults. The efficiency of existing spectrum-based approaches suffers from overlapping test cases describing different failures as well as coincidentally correct test cases which execute defects but do not verify their appearance. The selection of suitable test suites allows for ignoring such problematic tests and to focus on a single point of failure. As the collection of test cases plays an important role in how efficiently a fault is localized, we argue that especially unit testing frameworks assist developers in making a good choice [216]. Each test suite verifies a specific system part and is easily selectable in test runners. Furthermore, based on the origination condition of single faults [182], which means each failure must evaluate the defect, we allow developers to optionally filter methods which were not executed by all failing tests. If developers can ensure that they work on one failure, we hide all anomalies with a confidence score smaller than one because they have been covered only by a subset of all failing test cases. Consequently, developers can choose designated unit test suites, further reduce fault localization results, and concentrate on one specific failure at a time.

4.1.3. Refinement of Anomalies at Statement-level

We ensure scalability of spectrum-based fault localization by efficiently recording test coverage at methods and refining statement coverage on demand. So far, spectrum-based fault localization techniques record covered statements to provide fine-granular information about suspicious source code entities. However, the dynamic analysis of statement coverage tends to be slow and neighboring program entities often have identical anomalous scores because they are executed in sequence. Therefore, current spectrum-based approaches do not scale very well with large systems and provide too much information for an initial overview of the system. To limit the run-time overhead and to provide a compact visualization, we first collect spectrum-based anomalies at methods. Nevertheless, we also support the identification of failure causes in full detail by refining coverage information

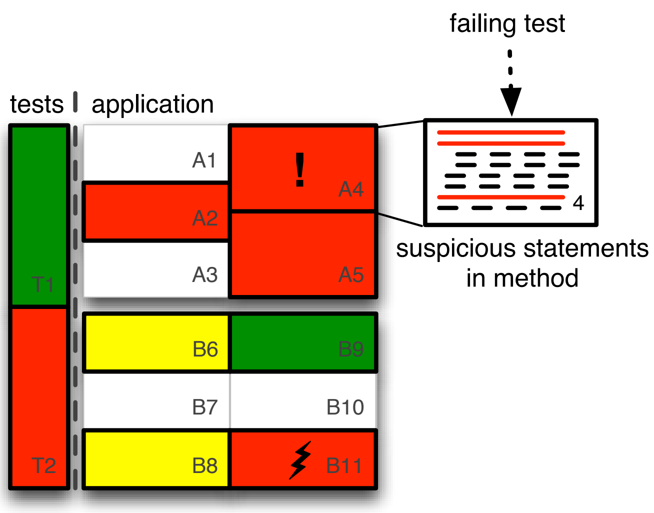


Figure 4.3.: We collect spectrum-based anomalies at methods first. If developers are interested, we refine the results at statements by executing covering tests again.

inside specific methods on demand. If developers request additional information for a method, we run all its covering tests, obtain the required statement coverage, and compute spectrum-based anomalies with the same formulas as for methods. This approach restricts the performance decrease of statement-level analysis only to methods of interest. Developers have fast access to anomalous methods and optional refinements for all details of suspicious statements. Combining spectrum-based fault localization and our refined coverage analysis provides a good trade-off between performance and fault localization details. Figure 4.3 illustrates our optional refinement of suspicious statements. Only after developers are interested in the details of the very suspicious and complex method 4, we re-execute its failing test and collect its covered statements. With this information, only this specific method computes spectrum-based anomalies and highlights the suspicious statements inside.

4.1.4. Example: Localizing Suspicious Response Objects

In our motivating typing error, we localize several anomalies within Seaside's response methods. Figure 4.4 presents the tree map visualization of Seaside[2] 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 colors methods with their suspiciousness scores 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

[2]For the purpose of clarity, we limit the system under observation to Seaside's core packages.

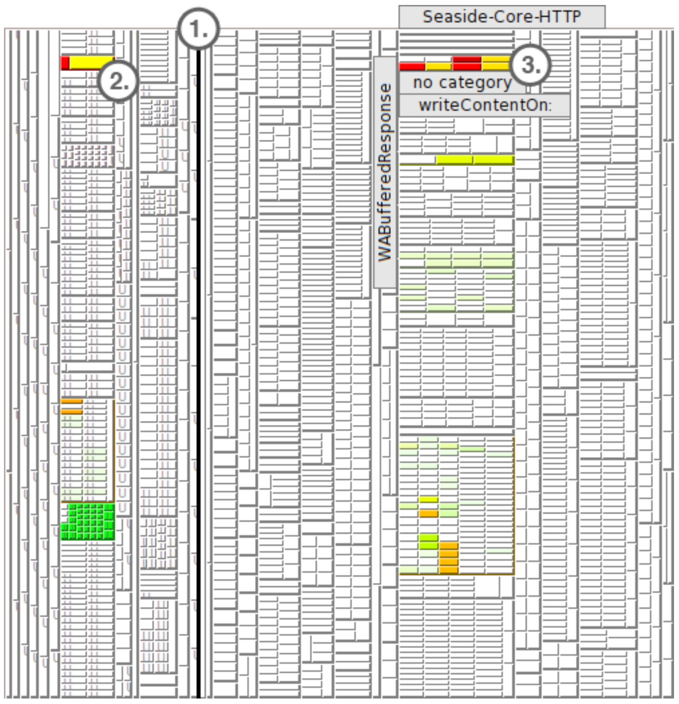


Figure 4.4.: In our Seaside example, our structure navigation restricts the search space to a few very suspicious methods in buffered responses.

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 share 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 additional help in order to understand how the failure comes to be.

4.2. Team Navigation: Recommending Experienced Developers

As understanding anomalies and failure causes requires thorough familiarity with suspicious system parts, our team navigation proposes a new metric for identifying expert knowledge even if defects are still unknown. More experienced developers tend to invent better hypotheses [11] and so require less time for debugging compared to novices [88]. Typically, in large projects where one developer cannot keep track of all program details, an important task is to find experts that are likely able to explain erroneous behavior or even fix the defect itself. Assuming that the author of the still unknown defect is the most qualified contact person, we restrict the search space to suspicious system parts and approximate developers that have recently worked on corresponding methods.

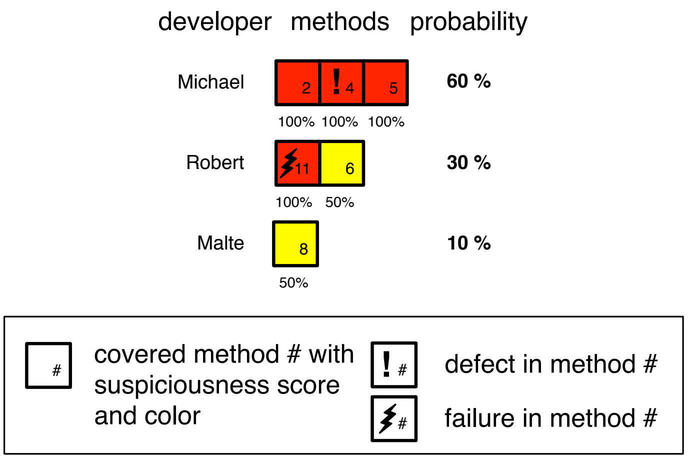


Figure 4.5.: Our team navigation sums up all authors of spectrum-based anomalies and recommends a list of experienced developers.

Consequently, as anomalies have a high probability to include failure causes [122], our anomaly-based metric recommends developers that comprehensively understand infections or possibly the defect itself. Figure 4.5 summarizes our developer ranking metric and its relationship to spectrum-based anomalies of our structure navigation. To calculate the developer ranking, 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. As Michael has developed the most suspicious methods (2, 4, and 5), he tends to be an expert for debugging this failure. He causes not only the defect in method 4 but also understands large parts of the infection chain. Even if Robert is responsible for the observable failure in method 11, Michael fits better as an expert because he is able to explain failure causes entirely.

4.2.1. Anomaly-based Developer Ranking

Our team navigation proposes a novel developer ranking metric that restricts the search space for expert knowledge to spectrum-based anomalies. First, we extrapolate a new set of methods from the system under observation (M_{System}) which includes those also identified by spectrum-based fault localization.

$$M_{Suspiciousness} = \{m \in M_{System} \mid suspiciousnessOf(m) > 0\}$$

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 most active author. The function *authorOf()* is independent of our approach and can be replaced by arbitrary heuristics that return expert knowledge for a specific method such as most activity, last access, and initial implementation.

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

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 and vice versa [205].

$$FMeasure(M) = 2 \cdot \frac{\left(\sum\limits_{m \in M} suspiciousnessOf(m)\right) \cdot \left(\sum\limits_{m \in M} confidenceOf(m)\right)}{\sum\limits_{m \in M} suspiciousnessOf(m) + confidenceOf(m)}$$

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

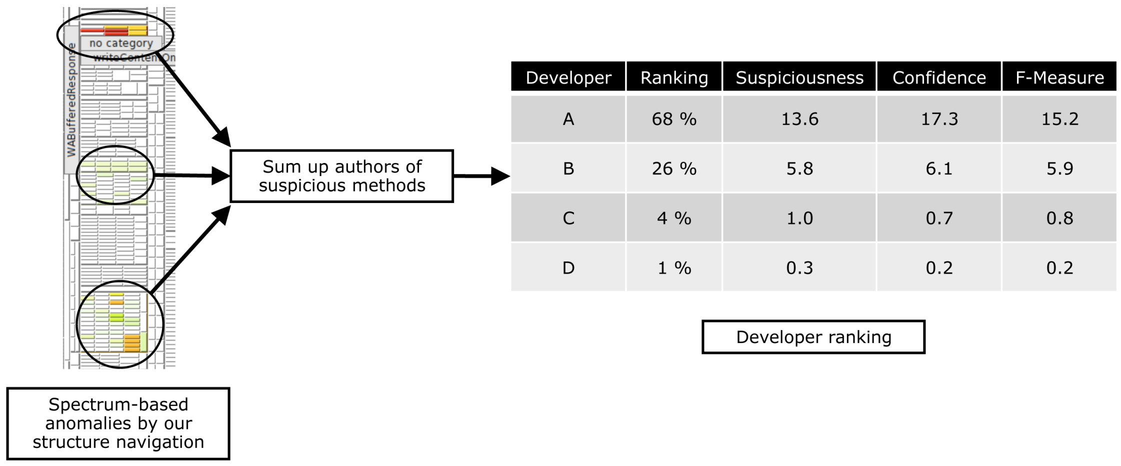
$$developerRanking(Developer) = \frac{FMeasure(M_{Developer})}{FMeasure(M_{Suspiciousness})}$$

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

4.2.2. Example: 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 (Developer A[3]) is marked as particularly important. The table in Figure 4.6 summarizes the (interim) results of the developer ranking metric and suggests Developer A 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 would be shrinking (only 55 %), C (24 %) changes 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 cost of debugging.

[3]Developers' names have been anonymized.


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 4.6.: Our developer ranking classifies (anonymized) experts. After analyzing the authors of spectrum-based anomalies, a ranked list presents possible experts that understand failure causes best.

4.3. Behavior Navigation: Debugging Erroneous Test Cases Backwards

To understand what happens before failures, our behavior navigation offers both fast access to execution histories and guidance along suspicious run-time data. Developers can experiment and observe the entire infection chain of a failing test case and systematically follow it back to its root cause. Based on the idea of test cases as reproducible and deterministic entry points, we split the expensive dynamic analysis over multiple test runs. Depending on developers' needs, we refine the execution trace step by step and so ensure an experience of immediacy when exploring behavior. In addition to it, we reuse spectrum-based anomalies to classify the large amount of trace data and so assist developers in tracking down failure causes. Thus, we provide back-in-time capabilities for failing unit tests with a special focus on fault localization.

The integration of anomalies into execution traces solves the shortcomings of spectrum-based fault localization and back-in-time debugging. As previously mentioned, isolated anomalies provide only unrelated starting points into debugging because the structural overview cannot relate them to failure causes. We argue that developers require an additional view to reveal how suspicious methods and malicious behavior belong together. Such a view is especially supported by back-in-time debuggers as they can map and arrange anomalies along execution histories. On the other side, back-in-time debugging produces large and confusing execution traces. If developers follow the infection chain step by step, they have to make countless decisions on how to follow corrupted behavior and state. To reduce the effort required, spectrum-based anomalies can highlight infection chains in order to run like a common thread through the large amount of run-time data. By combining anomalies and execution traces, our behavior navigation assists developers in both understanding anomalies and following failure causes back.

4.3.1. Following the Infection Chain Backwards

For the purpose of localizing failure causes, we offer developers access to the entire execution history of a failing test case. Starting with a test case as a reproducible entry point, we record its behavior, present the run-time data in form of an explorable call tree, and so allow developers to follow the infection chain back to its root cause. In doing so, we provide arbitrary navigation through method call trees and their state spaces in both forward and backward direction. We restrict dynamic analysis to partial traces and, primarily, to the granularity level of methods [91]. Apart from common back-in-time features, such as a query engine for getting a deeper understanding of what happened, our approach possesses two distinguishing characteristics. First, our incremental dynamic analysis allows for immediate access to run-time information of reproducible and deterministic test cases. Second, the integration of anomalies classifies suspicious trace data and so facilitates navigation in large traces.

We ensure an immediate debugging experience when exploring behavior by splitting run-time analysis of test cases over multiple runs [168]. 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* as part of our incremental dynamic 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 user-relevant details (state of variables, profiling data, statement coverage) on demand. Thereby, test cases fulfill the requirement to reproduce arbitrary points of a program execution in a short time [197]. Thus, by distributing dynamic analysis cost into multiple test runs, we ensure quick access to relevant run-time information without collecting needless data up front.

4.3.2. Anomalous Behavior Guides Developers to Defects

We classify behavior with respect to suspiciousness scores of methods for an efficient localization of failure causes in large traces. To divide the trace into more or less erroneous behavior, we either reuse the already ranked methods of our structure navigation or rerun the spectrum-based fault localization on traced methods again. On the analogy of our structure navigation, we color the trace with suspiciousness and confidence scores at each executed method. Moreover, a query mechanism supports the navigation to erroneous behavior. We expect that developers identify failure causes in our classified traces more quickly because they allow abbreviations to methods that are likely to include failure causes. As a result, we support developers during hypothesis-testing and their laborious decisions on how to follow infection chains backwards.

Figure 4.7 illustrates the integration of anomalies into execution histories of failing test cases. The infection chain is classified with anomalies and developers can directly start

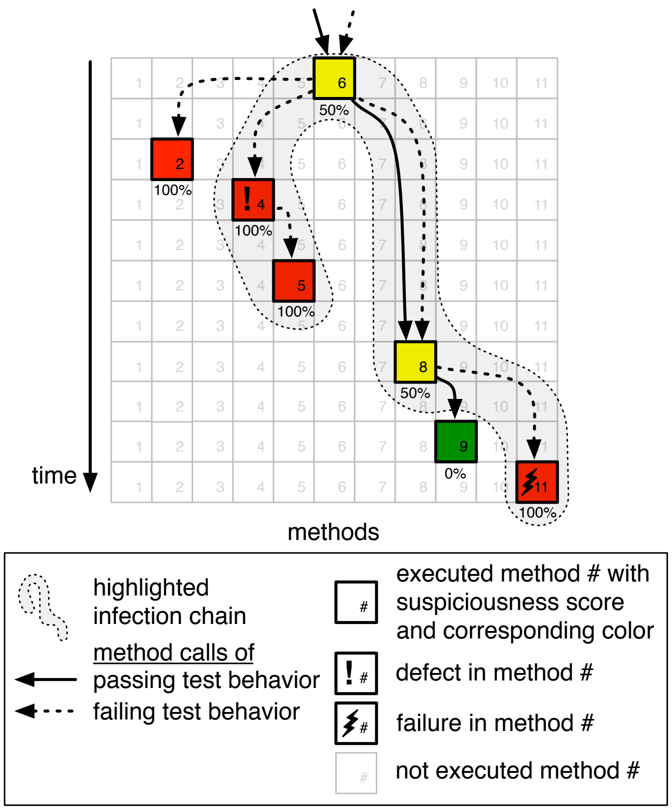


Figure 4.7.: Our behavior navigation gives developers helpful advice on how to follow the infection chain backwards. The execution history includes information about failure cause probabilities at each single method so that developers can decide how anomalies are related to each other and where to focus debugging.

debugging on the left sub tree that forms a suspicious behavior. As methods 8 and 6 are less suspicious, developers can shorten the infection chain to methods 5, 4, and 2. Furthermore, as method 2 is called before the defect in method 4, developers following erroneous behavior backwards do not need to check this anomaly because they have already found the root cause. So, we support the understanding on how suspicious structure and behavior belong together.

In addition to the classification of execution histories, anomalies also help in choosing the plainest entry point if there are several failing test cases available. Often a defect causes more than one test case to fail, which means developers have to decide by their own which execution history to follow. Without feedback about test case behavior, it is pure coincidence if they choose the shortest or largest infection chain. As all failing test cases include the defect, it is important to identify the plainest entry point because a shorter infection chain also implies less effort during debugging. For that reason, a new metric that estimates the infection chain length per failing test case is preferable.

$$InfectionLengthOfTest(M_{Test}) = \sum_{m \in M_{Test}} suspiciousnessOf(m)$$

For each failing test case, we compute its covering methods and sum up their suspiciousness scores. In doing so, we assume that failing test cases with fewer anomalies have shorter infection chains than test cases that cover numerous suspicious methods until they fail. In other words, the lower the value of our metric, the shorter the infection chain and the required debugging effort. We sort all test cases in ascending order so that we rank the test case with the lowest number of anomalies first. This test case should be preferred to start debugging.

4.3.3. Example: Understanding How the Failure Comes to Be

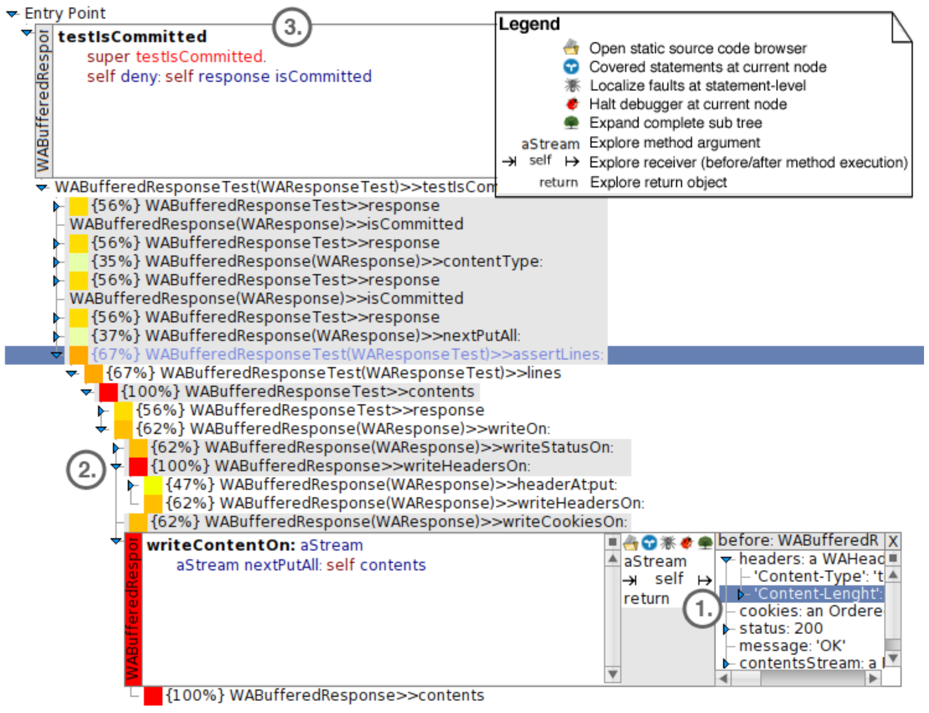
In our Seaside example, we highlight the erroneous execution history of creating buffered responses and support developers in understanding how suspicious methods belong together. Following Figure 4.8, developers focus on the failing `testIsCommitted` behavior and follow the shortest infection chain 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 state and behavior first[4]. A detailed inspection of the receiver object reveals that the typing error 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)[5]. Considering `writeHeadersOn:` manifests the failure cause in the same way. 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: Identifying Corrupted State in Infection Chains

While our behavior navigation highlights suspicious and executed methods, our state navigation further identifies corrupted state in the infection chain. To detect state anomalies, we first analyze the hidden test knowledge of passing tests, then create contracts with their common object properties, and finally compare them with failing test case paths. Differences between concrete and common state reveal anomalies that have

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

[5]If developers follow the flow of the corrupted receiver object step by step, they also have to check all methods in between.



```
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

```
before: WABufferedR
  headers: a WAHead
    'Content-Type': 't
    'Content-Lenght':
  cookies: an Ordere
  status: 200
  message: 'OK'
  contentsStream: a
```

Figure 4.8.: The classified execution history of our Seaside typing error.

a high probability to include failure causes. The entire process of our state navigation works automatically and only requires a comprehensive test suite with several passing test cases. Analogous to our spectrum-based anomalies, we embed the uncovered state anomalies into the execution history and so highlight how corrupted state is related to each other and the infection chain. As a result, we further restrict the search space and assist developers in observing and experimenting with failure causes. With respect to the scientific method, our state navigation provides another valuable opportunity to refine hypotheses and to make conclusions about root causes.

4.4.1. Harvesting Likely Invariants from Passing Test Cases

We start our state navigation by revealing the hidden test knowledge and harvesting invariants. To learn common object properties, we first identify all passing test cases, execute them, and collect their used objects. For each method called, we check its arguments, the return value, and the receiver object. Having a concrete object, we accumulate its specific properties into generalized invariants that hold with previously observed objects at the same program entity. We analyze several properties that are important for program comprehension from type information to value ranges of numbers, collections, and strings. During the execution of this inductive analysis, we steadily

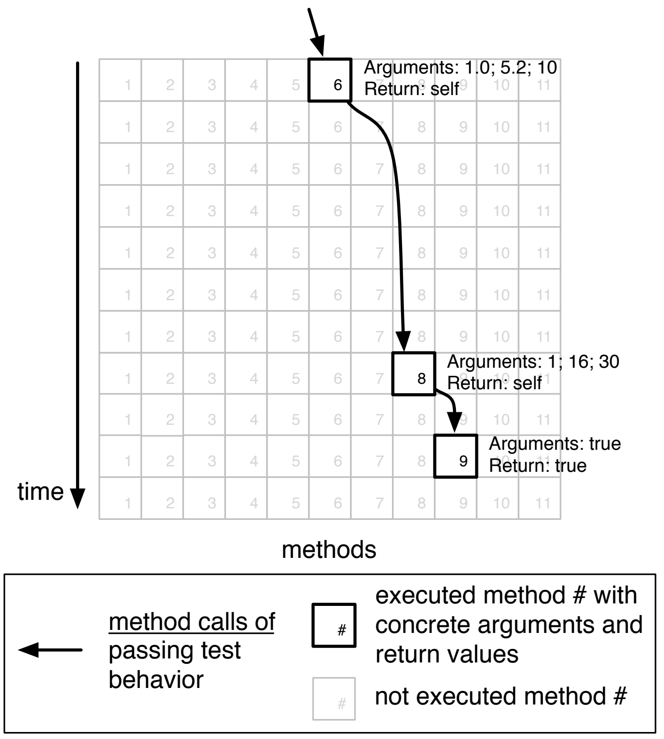


Figure 4.9.: The hidden test knowledge of passing test runs forms with its concrete values implicit invariants of the system under observation.

expand likely invariants once concrete objects come along with new properties that are not covered so far. For example, an instance variable contains both integer and float objects in different executions. In this case, we first collect an integer type as likely invariant. Later, when the float object occurs, we expand the likely invariant to the number type as their common super class. By harvesting concrete objects during the execution of test cases, we gather enough data for pre- and post-conditions of methods as well as invariants of accessed instance variables that support developers not only in understanding software.

In Figure 4.9, we illustrate the concrete objects of a passing test case example. In different executions, method 6 is called with 1.0, 5.2, and 10 at the same argument. During the first method call, we collect a float type in the value range of 1.0 to 1.0. With the second execution, we extend the value range to 5.2. With the last integer argument, we change the type to the super class `Number` and enlarge the value range again. Thus, we have a generalized object property for this argument that expects numbers between 1 and 10. We also harvest likely invariants by all other called method, their arguments, return values, and receiver objects. Method 8 allows integer arguments between 1 and 30 and always returns a receiver object with the same specific class type. All observable objects of method 9 are the same true booleans and for that reason they uncover a possibility for writing new tests.

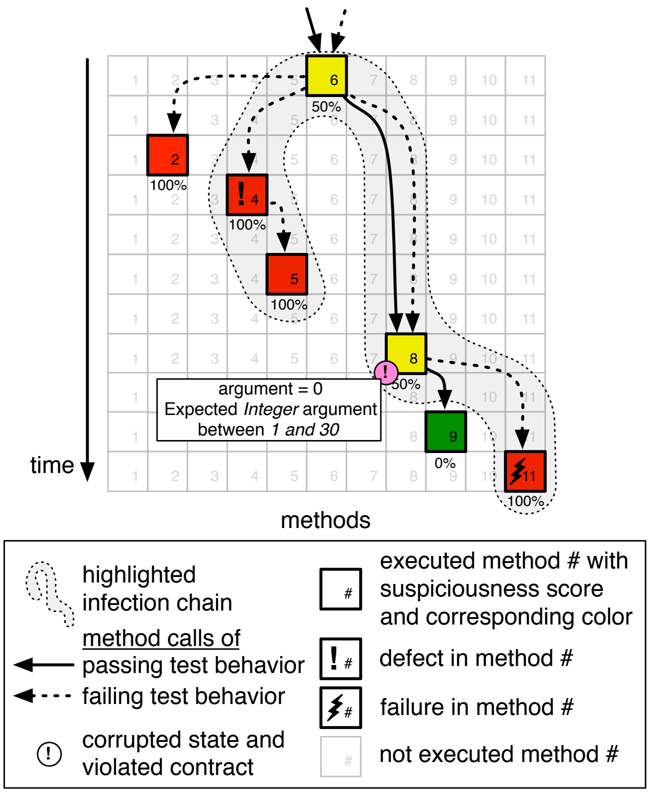
Although the collected information is helpful with respect to program comprehension and later debugging, its quality strongly depends on the test base. We assume that concrete objects have meaningful values and that tests check the entire code base comprehensively. The first assumption is valid because test cases check all kinds of border cases and so they include the most important information for program comprehension. Our harvesting tries to recover these implicit cases in form of likely invariants that represent equivalence classes of testing approaches. The second assumption sometimes has limitations when specific methods are executed one time only. If test cases are not extensive enough, our invariants tend to be too specific and close to concrete objects. Nevertheless, we argue that this information is still a valuable source for understanding the system even if the run-time data does not include large value ranges.

4.4.2. Detection of State Violations with Dynamic Contracts

With harvested invariants, our state navigation generates dynamic contracts that automatically detect corrupted state in failing test cases. For each generalized object property, we create suitable assertions and aggregate them into contracts. Depending on the covered program entity, we add contracts for arguments as pre-conditions to methods, return values as post-conditions to methods, and receiver state as invariants to their corresponding classes. With the help of these assertions, developers are able to run test cases with activated contracts and check them for state anomalies. While passing test cases should work correctly, failing test cases tend to create violations if their state is not in accordance to our harvested and correct invariants. Such differences between expected and anomalous state have a high probability to include failure causes so that developers can rely on this information for creating their hypotheses. To further reveal parts of the infection chain and to understand how anomalies relate to each other, we map these state violations again to the execution history. Thus, developers can experiment and observe the failing test case behavior and our state navigation guides them along the infection chain.

In Figure 4.10, we present how dynamic contracts are able to detect state anomalies. Based on the harvested invariants, our failing test case calls method 8 with 0 as argument and violates the contract. While the type assertion is still valid, the value range assertion allows only numbers between 1 and 30. This state anomaly is a valuable indication for a hypothesis and with its mapping on the execution history, it further reveals a part of the infection chain. Developers are able to understand the corrupted state and its origin.

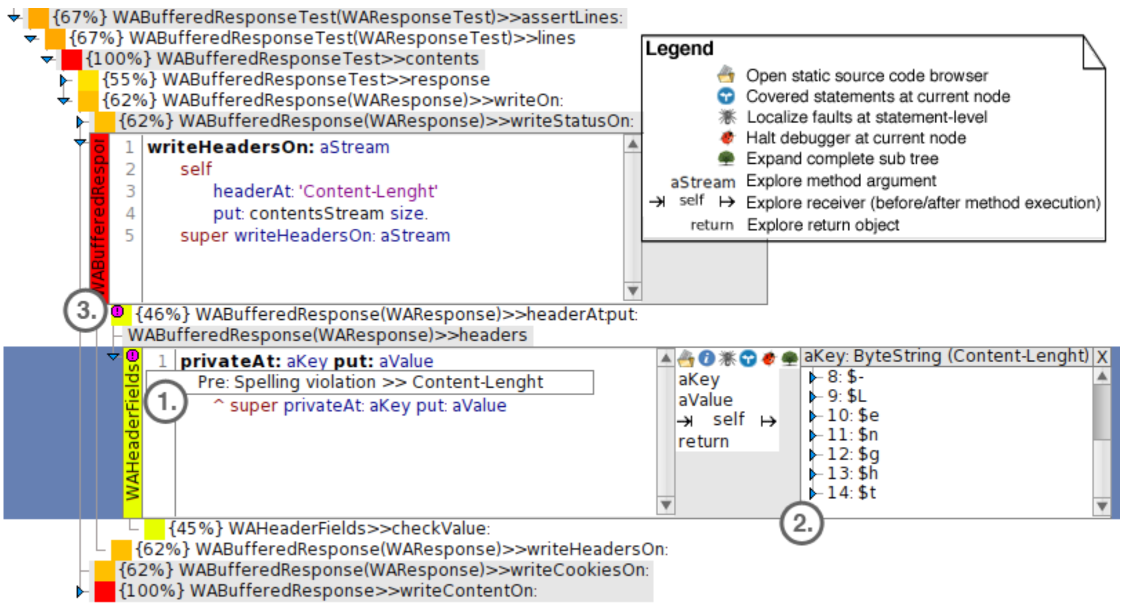
Our dynamic contracts support the classic design by contract [151] and extend it with some specific features that are required for our state navigation [106]. We provide invariants for a class and pre- and post-conditions for methods. *Invariants* belong to a class and hold when an object is in a valid state. We evaluate invariants before and after every call to a (public) method. *Preconditions* ensure that the input values for a method have the right properties and the invariant holds. *Postconditions* ensure that



time
methods
argument = 0
Expected *Integer* argument between *1 and 30*
50%
100%
0%
highlighted infection chain
method calls of
passing test behavior
failing test behavior
corrupted state and violated contract
executed method # with suspiciousness score and corresponding color
defect in method #
failure in method #
not executed method #

Figure 4.10.: Differences between the derived invariants and failing test cases reveal state anomalies that are likely to include failure causes.

the return values have the right properties and that the receiver object is still in a valid state after returning to its caller. This also includes the comparison of *old* values, representing state before the method was executed, with newly computed values after the execution. In addition to these features, we extend the flexibility of current design by contract approaches in three ways. *Grouping* contracts and their selective activation keep things together and reduce the required performance overhead. Instead of checking the entire system, developers can apply contract groups to only verify system parts that have something to do with the observed failure. Criteria for forming such groups are up to developers, but we offer pre-built groups for packages and all kinds of assertions. *Scoping* only activates contracts for the local thread of execution. This is especially useful while debugging production systems, where developers test system behavior without influencing the execution flow of concurrent clients. We limit the scope of our dynamic contracts to processes started from test runners. *Dynamic contract enforcement* facilitates contract activation and deactivation at run-time. There is no need to recompile the entire system in order to switch contracts on or off. Due to these three features we are able to limit the performance overhead of checking contracts to suspicious methods. Other methods with dynamic contracts run with nearly full speed because we check the observed group, scope, or enforcement property at method activation only once.



```
{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
        3.
          {46%} WABufferedResponse(WAResponse)>>headerAt:put:
          WABufferedResponse(WAResponse)>>headers
            WAHeaderFields
            privateAt: aKey put: aValue
              Pre: Spelling violation >> Content-Lenght
            1.
              ^ super privateAt: aKey put: aValue
            aKey
            aValue
            self
            return
            aKey: ByteString (Content-Lenght)
              8: $-
              9: $L
              10: $e
              11: $n
              12: $g
              13: $h
              14: $t
            2.
            {45%} WAHeaderFields>>checkValue:
          {62%} WABufferedResponse(WAResponse)>>writeHeadersOn:
        {62%} WABufferedResponse(WAResponse)>>writeCookiesOn:
        {100%} WABufferedResponse>>writeContentOn:
```

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 4.11.: State anomalies highlight the typing error and reveal the infection chain near the defect.

4.4.3. Example: Coming Closer to the Typing Error

In our Seaside typing error, our state navigation is able to reveal two anomalies close by the root cause. By analyzing all passing tests from the still working streamed responses, we are able to collect type and value ranges of all applied objects. Among others, we check whether all string objects are spelled correctly or not. After that, we derive common invariants from the concrete objects and create corresponding contracts. 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. Lastly, we execute the same failing test case as in our behavior navigation, but now with enabled contracts. As soon as a contract is violated, we mark the corresponding exception in the execution history and so reveal two state anomalies for our `testIsCommitted` that are close to the defect.

Figure 4.11 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 typing error 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 caused the failure.

4.5. Summary

We explained the four specific techniques of our test-driven fault navigation and its relation to the scientific method in more detail. Structure navigation localizes suspicious system parts and so supports the creation of hypotheses. It emphasizes relationships between spectrum-based anomalies and provides an overview of starting points that are likely to failure causes. Team navigation recommends other developers for helping with creating, predicting, and later refining hypotheses. We restrict the search space to authors of suspicious program entities only and suggest suitable experts even if the defect is still unknown. Behavior navigation allows developers to experiment with the entire execution history and to explore arbitrary object states. With the help of anomalies, it further classifies erroneous behavior for facilitating the navigation through the large amount of run-time data. State navigation reveals parts of the infection chain and assists in the observation of and conclusion about failure causes. After harvesting invariants of passing test cases, dynamically created contracts violate failing test cases and uncover state anomalies in the infection chain. With all four navigations, developers have a systematic and comprehensive approach for debugging failures back to their root causes.

5 Incremental Dynamic Analysis

In this chapter, we introduce our incremental dynamic analysis which provides the required run-time data for our test-driven fault navigation in a short amount of time. We start with the foundations of our approach and how it ensures an experience of immediacy when debugging later with our tools (Section 5.1). After that, we present our three specific incremental dynamic analysis techniques in more detail. For structure navigation, our refined coverage analysis provides fast access to method coverage and optionally refinements at statements (Section 5.2). For behavior navigation, our step-wise run-time analysis splits the expensive dynamic analysis of complete execution histories over multiple test runs (Section 5.3). For state navigation, our inductive analysis just harvests generalized object properties that developers need for their current debugging task (Section 5.4).

5.1. Immediacy through Interactivity

Traditional dynamic analysis techniques such as post-mortem debuggers [135] are typically inefficient and time-consuming. Most approaches are designed for general analysis purposes and thus they capture comprehensive information about the entire execution up-front [56]. Unfortunately, the required run-time analysis is often associated with an inconvenient overhead that renders their current tools impractical for frequent use [58].

We argue that the overhead imposed by current approaches to dynamic analysis is uncalled-for and that immediate accessibility of run-time information is beneficial to developers. Continuous and effortless access to run-time views supports developers in acquiring and evaluating their understanding for debugging. For that reason, the overhead caused by dynamic analysis has to be minimal. If analyzing test behavior is time-consuming, developers will either not run them very often or reject the entire approach.

Debugging tools that provide run-time information require an experience of immediacy to encourage their frequent use [168, 201]. To that effect, two essential characteristics need to be met. First, debugging tools have to be integral parts of the programming environment. Developers would welcome a tool carrying them from method source code to the actual run of the same method interactively. Second, response times have to be low. Visualized run-time information has to be available within some hundreds of milliseconds rather than minutes [190]. However, immediacy must not hamper the level of detail available from views. We intend to support debugging by reducing the effort of accessing run-time information. We aim to encourage developers to use our test-driven fault navigation approach frequently. Developers shall be able to avoid guesswork and validate assumptions by inspecting actual run-time information *immediately*.

We employ a new approach to dynamic analysis that enables an experience of immediacy that current tools are missing. Our incremental dynamic analysis is an interactive and incremental approach to collect and present run-time data. Low cost can be achieved by structuring program analysis according to user interaction. More specifically, user interaction allows for dividing the analysis into multiple steps: A high-level analysis followed by on-demand refinements. An initial analysis provides for immediate access to visualizations of run-time information. As users explore this information, it is incrementally refined on demand. This distinction reduces the overhead to provide run-time information while preserving instantaneous access to detailed data. Splitting the dynamic analysis over multiple runs is meaningful because developers typically follow a systematic approach to understand program behavior. More generally, program comprehension is often tackled by exploring an overview of all run-time information, and continuing to inspect details [209]. This systematic approach to program comprehension guides our incremental dynamic analysis: Run-time data is captured when needed.

This interactive approach to dynamic analysis requires the ability to reproduce arbitrary points in a program execution. In order to refine run-time information in additional runs, we assume the existence of entry points that specify deterministic program executions. For our implementation, we leverage test cases as such entry points because they commonly satisfy this requirement [149]. However, our incremental dynamic analysis is applicable to all entry points that describe reproducible behavior.

We analyze the execution of entry points by instrumenting the code using flexible method wrappers [40]. Method wrappers are lightweight first-class entities that transparently replace methods with alternative implementations and can delegate to the original (wrapped) implementation. They can be handled like ordinary objects, allowing for simple extensions of method behavior as well as dynamic installation and deinstallation. We use different kinds of wrappers to decorate methods with additional functionality required during recording run-time data at different levels of detail. Moreover, we restrict instrumentation of application code to relevant methods. Library and framework methods being of no interest are excluded from wrapping, yielding *partial traces*. The selection of relevant packages and exclusion of others further avoids unnecessary overhead [91].

Based on the idea of distributing the dynamic analysis effort across multiple runs, our incremental dynamic analysis offers three specific techniques for our test-driven fault navigation to collect coverage, execution histories, and object properties on demand. *Refined coverage analysis* provides test coverage at methods and optional refinements at statements. In doing so, our structure navigation owns both fast access to suspicious system parts and if desired full details of anomalous source code. *Step-wise run-time analysis* first collects an initial call tree and allows developers to refine further details later

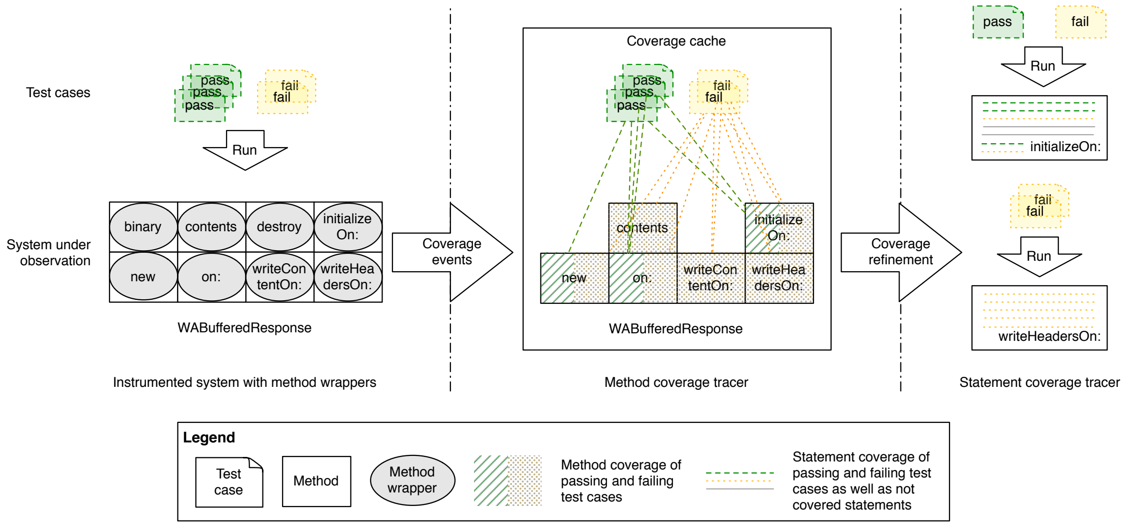


Figure 5.1.: Our refined coverage analysis first collects method coverage and later refines results at statements on demand.

on. For our behavior navigation, developers immediately retrieve a shallow overview of the execution history that is increasingly expanded with user-relevant run-time data. *Inductive analysis* splits the expensive recording of all object properties to specific debugging needs. In state navigation, developers do not record all possible objects at once but rather choose from several harvesting types and complete missing data in succeeding runs.

5.2. Refined Coverage Analysis

We collect coverage information at different levels of detail to ensure scalability for our structure navigation and its spectrum-based fault localization. Although the pure collection of statement-level coverage delivers the most detailed results, their analysis often includes an unjustified overhead. The dynamic analysis slows down the execution by a factor of up to 100 and so requires too much time for revealing anomalies [103]. Moreover, we argue that object-oriented method implementations often include only sequences so that their covered statements are equal to entirely covered methods. For these reasons, we restrict the performance decrease to methods and only collect statement coverage on demand. Starting with a complete but fast analysis of method coverage, we reveal the relationship between test cases and their executed methods. After that, if required, developers can refine statement coverage of specific methods. Based on reproducible test cases, we re-execute all tests that execute this method and analyze the covered statements only for the method of interest. In doing so, we offer developers both fast access to method coverage and optionally refined statement coverage.

Figure 5.1 summarizes our refined coverage analysis with a subset of our Seaside typing error example. We incrementally analyze coverage of the most suspicious `WABufferedResponse` class, its methods, and selected statements. We split the dynamic analysis into a fast collection of method coverage and subsequent refinements of statement coverage of specific methods:

Method coverage reveals the relationship between test cases and their executed methods of the system under observation. The first part of our refined coverage analysis starts with wrapping all application methods. In doing so, we limit the analysis scope to the parts of the system in which developers are interested in. So, we filter irrelevant methods and restrict the performance decrease for the dynamic analysis in advance. For example, in Figure 5.1 we limit the coverage scope to our `WABufferedResponse` class and its eight methods only. Having an instrumented system, we run each test case on its own. As soon as a wrapped method is called, we send a coverage event to our method tracer that then stores the link between the test case and the executed method. Having all covered events of a specific test, we store all coverage relationships, cache the final test result, and make them available to any interested tool. In Figure 5.1, we cover six out of eight methods after executing all test cases. Two methods have not been executed at all, three methods are covered by both kinds of test results, and the last three methods are only related to failing test cases. Based on this data, our structure navigation can already compute and present the suspicious system parts that help developers to restrict the initial search space.

Statement coverage allows developers to optionally refine covered methods at the source code level. To compute on-demand statement-level coverage for a specific method, we wrap it with a special simulation wrapper and rerun only its covering tests in background. The simulation wrapper records covered statements by executing the method's byte code with Smalltalk's interpreter engine. As soon as this method is executed, we start the interpreter and store all executed byte codes which we can later map to covered statements. If the method implementation calls another method or returns, we suspend the interpreter and the system runs with full speed until we return to the wrapped method again. In Figure 5.1, we refine two methods. The first method has been covered by passing and failing tests and we can refine spectrum-based fault localization at statements. The second method is only executed by failing tests, thus, we only reveal that all statements are equally covered and consequently suspicious as the entire method.

Our refined coverage analysis allows developers fast access to initial results and full details on demand. During the method coverage, the performance only decreases by a factor of two on average, which we argue is usually less perceivable than a factor of 100 by a simulated interpreter. To obtain full details, it is the developer's decision to refine the coverage information step by step. As a consequence, we also prevent a large amount of unimportant coverage data. For these reasons, our refined coverage analysis also forms a basis for the scalability of spectrum-based fault localization. We allow fast access to anomalies at covered methods, collect only requested data, and still allow access to fine-granular suspicious statements. Thus, our on-demand coverage analysis provides a good trade-off between performance and the levels of detail (see Section 7.5 for detailed benchmarks).

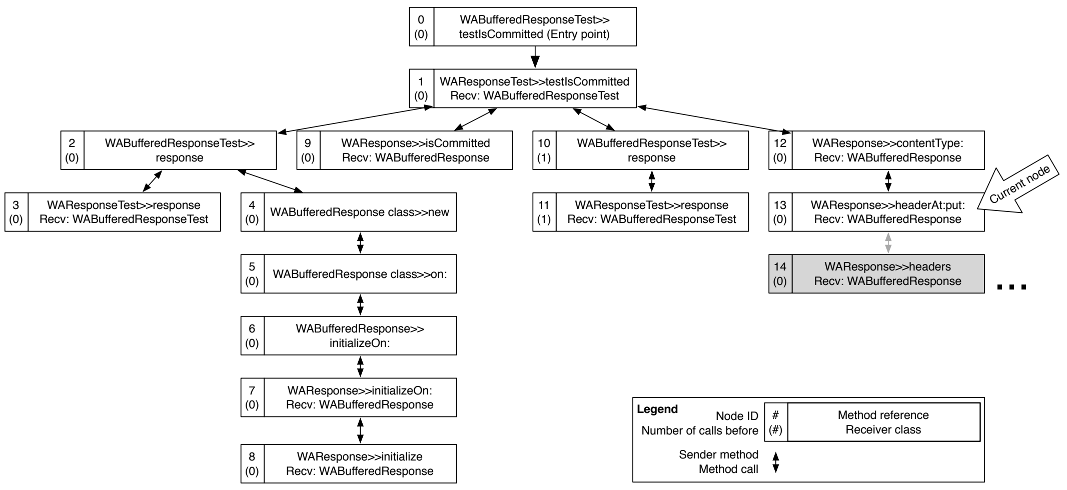
5.3. Step-wise Run-time Analysis

For our behavior navigation, we provide an incremental dynamic analysis technique that supports the exploration of complete execution histories of specific test cases. Our step-wise run-time analysis allows interactively access to a behavioral path through the system. Thus, developers are able to follow infection chains back to their root causes and entirely understand the problem. Our step-wise run-time analysis splits the exploration of a whole test case execution into an initial shallow analysis and detached refinement analyses:

Shallow analysis focuses on the information that is required for presenting an overview of a program's execution history. For example, method and receiver names are sufficient to render an initial call tree. Further information about method arguments or instance variables is not recorded. The amount of required data for generating an initial overview is limited compared to the information that is generated in an entire program run. More specifically, the overhead for collecting method name and receiver information is significantly less than performing a full analysis.

Refinement analysis records user-relevant details of the execution history in detached program runs. As developers interact with the initial call tree, they identify interest in individual objects which are then loaded on demand in additional analysis steps. Such subsequent refinement steps involve recording of object state at the specified point in the execution. For example, if developers indicate interest in a specific method argument, the refinement analysis executes the entry point again and only collects this object. Due to reproducible and deterministic entry points, we can ensure the same behavioral path and object space for each execution. Furthermore, a refinement step imposes a minimal overhead by focussing on a single object at a particular execution step. This means that refinement analysis is hardly more expensive than execution without instrumentation.

Our step-wise run-time analysis distributes the effort for dynamic analysis across multiple runs and allows developers to recover complete execution histories step by step. The information required for debugging is arguably a subset of what a full analysis of a program execution can provide. While our approach entails multiple runs, the additional effort is kept to a minimum, especially when compared to a full analysis that has no knowledge of which data is relevant to the user. We reduce the cost by loading information only when the user identifies interest. This provides for quick access to relevant run-time



0 (0) WABufferedResponseTest>> testIsCommitted (Entry point)
1 (0) WAResponseTest>>testIsCommitted Recv: WABufferedResponseTest
2 (0) WABufferedResponseTest>> response
9 (0) WAResponse>>isCommitted Recv: WABufferedResponse
10 (1) WABufferedResponseTest>> response
12 (0) WAResponse>>contentType: Recv: WABufferedResponse
3 (0) WAResponseTest>>response Recv: WABufferedResponseTest
4 (0) WABufferedResponse class>>new
11 (1) WAResponseTest>>response Recv: WABufferedResponseTest
13 (0) WAResponse>>headerAt:put: Recv: WABufferedResponse
Current node
5 (0) WABufferedResponse class>>on:
14 (0) WAResponse>>headers Recv: WABufferedResponse
...
6 (0) WABufferedResponse>> initializeOn:
7 (0) WAResponse>>initializeOn: Recv: WABufferedResponse
8 (0) WAResponse>>initialize Recv: WABufferedResponse
Legend
Node ID
Number of calls before
# (#)
Method reference
Receiver class
Sender method
Method call

Figure 5.2.: Building the initial call tree with our step-wise run-time analysis for our Seaside typing error.

information and execution histories without collecting needless data (see Section 7.5 for detailed benchmarks).

5.3.1. Shallow Analysis: Building the Call Tree

Shallow analysis ensures low start-up costs by building upon a lightweight call tree that is later on refined with additional information. In Figure 5.2, we sketch the initial call tree construction of our Seaside typing error. At first, we create a tracer object for the test case entry point in question. The tracer object is responsible for constructing and managing the call tree. During the execution of entry points, *method call wrappers* signalize events that lead to new nodes being inserted into the call tree. Initially, the call tree consists of a sole root node (with ID 0) representing the test case entry point (`WABufferedResponseTest»testIsCommitted`). All subsequent nodes are attached to the root or its children.

Next, we instrument all methods of relevant packages as specified by developers with method call wrappers. These reference the aforementioned tracer object and report collected data to it in the form of tracing events, which they signal as soon as each wrapped method is invoked. Now, the entry point is executed, leading to the production of tracing events and their consumption by the tracer object. The information carried by tracing events consists of the current method reference, the number of already traced calls to that method, and optionally the receiver's type. The latter is only included if it differs from that of the class declaring the method. The number of previous calls is relevant for later refinement analyses. In Figure 5.2, the tracer has already constructed the call tree up to the `headerAt:put:` method. The `WAResponse»headers` method is currently being run and the corresponding event consumed by the tracer.

From each tracing event, a new call node with a consecutive unique ID is created and inserted into the call tree. The relationship between a node in the tree and its children is bidirectional; downward links denote method calls, while upward links denote senders. To facilitate quick node insertion, the tracer object maintains a reference to the most recently inserted call node, called the "current node". A new node is inserted below the current node, and the current node is updated to reference the newly inserted node after that. This way, call node insertion takes place in constant time. In the figure, a new call node (with ID 14 and grey background) has just been inserted below the current node (ID 13). The new node represents the method `headers` in `WAResponse`, which has not been called before (the call counter is 0).

When methods terminate, the tracer is notified by the corresponding wrappers and reacts by adjusting the current node reference accordingly. The unique call node IDs are used to correctly drive this adjustment in case of recursive method calls and non-local returns. Once the entry point itself terminates, the shallow analysis is completed and all wrappers related to the given entry point are deinstalled.

5.3.2. Refinement Analysis: Refining Call Nodes

Our refinement analysis guarantees fast lookup and re-execution of entry points during which call tree nodes are augmented with deep copies of relevant objects. Once a call node for which details are requested is known, the method whose activation is represented by the node is instrumented with an *explore wrapper*. Explore wrappers, like call wrappers, signal tracing events to the tracer object; however, their payload consists of deep copies of relevant objects such as the receiver and method call arguments. Next, the tracer executes the entry point containing the call node in question. Since the explore wrapper wraps an entire method, it will be triggered multiple times during call tree re-execution. It must, however, produce tracing events *only* when the activation corresponding to the call node in question is met. This is realized by means of the count of already traced calls to the respective method: the wrapper maintains an internal activation counter, checking whether its value matches that of the call node's counter. As soon as the desired activation is recognized, the required deep copies are created and attached to the call node.

When refinement affects an object for the first time, a very deep copy is created right away[1], even though not all depth levels of the object's structure are of interest at this point in time. While this approach is arguably more memory-consuming than necessary, it is straightforward to implement. Investigation of a more fine-grained solution that copies object elements only as needed is deferred to future work. Furthermore, it should be noted that the structure of the call tree itself does not change at all during refinement analysis—it is merely augmented with new data. The wrapper is deinstalled once the requested information has been delivered.

[1]We rely on Smalltalk's `veryDeepCopy` method that automatically duplicates an entire object tree.

In the example in Figure 5.2, we assume that developers request additional information about the receiver object for the node with ID 10. In this case, we decorate the `WABufferedResponseTest»response` method with an explore wrapper and trigger re-execution at the test case entry point node (ID 0). Re-execution of the wrapped method at node 2 will not lead to augmentation because we are looking for the second method call. Reaching node 10 triggers deep copying of the relevant receiver object and adding it to the corresponding call tree node.

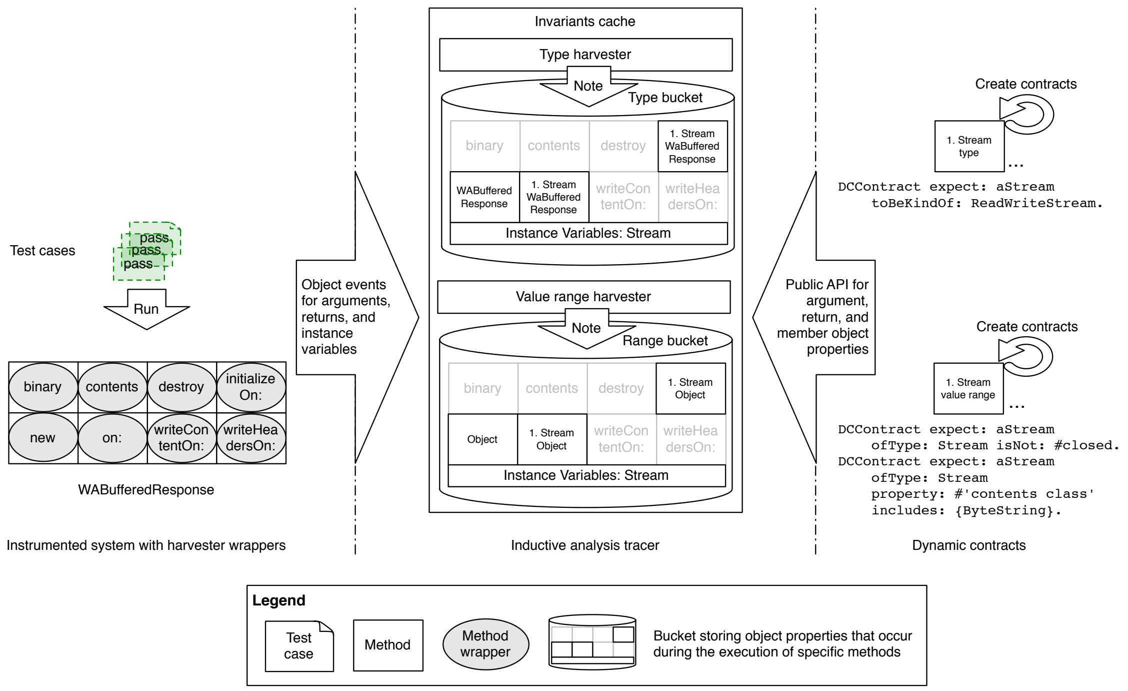
5.4. Inductive Analysis

Our inductive analysis harvests common object properties from passing test cases for our state navigation. With the help of these properties, we are able to create dynamic contracts that compare objects of failing test runs and so reveal state anomalies as parts of infection chains. However, the dynamic analysis of all object properties tends to be time-consuming [70]. During the analysis, we explore each object at each execution point in full detail in order to detect likely invariants. In large systems this method is not only expensive but also results in a vast amount of data that is not always relevant for a specific debugging task. For these reasons, our inductive analysis allows developers to collect object properties incrementally. We analyze not all data at once but rather developers choices depend on what interests them. They select the system under observation and decide about the specific object properties that should be harvested. If desired, they can refine the results later on by focusing on other object properties and re-executing our reproducible test cases again. Thus, we split our inductive analysis over multiple runs and provide selected results in a short time (see Section 7.5 for detailed benchmarks).

Figure 5.3 summarizes our inductive analysis approach and how method wrappers, the tracer with its harvesters, and dynamic contracts relate to each other.

On the left, developers first define the system parts from which they want generalized object properties. Based on this information, we instrument each of the selected classes and their methods with *harvester wrappers*. After that, we execute all passing test cases that cover the system under observation. If a wrapped method is executed, we send object events to our inductive analysis tracer. These events include the concrete run-time objects that this method receives as arguments, applies as instance variables, or returns as result. In the example of Figure 5.3, developers are only interested in generalized object properties of the suspicious `WABufferedResponse` class and its methods. If a passing test case calls the wrapped method `initializeOn:`, we send the concrete `ReadWriteStream` object as its first argument to our inductive analysis tracer.

In the middle, our inductive analysis tracer receives numerous object events and forwards them to its *harvesters* that derive generalized object properties. Depending on the developer's choice, our tracer includes several harvesters for different properties. Each harvester receives object events from the tracer and then generalizes specific object



```
DCContract expect: aStream
    toBeKindOf: ReadWriteStream.
```

```
DCContract expect: aStream
    ofType: Stream isNot: #closed.
DCContract expect: aStream
    ofType: Stream
    property: #'contents class'
    includes: {ByteString}.
```

Figure 5.3.: Our inductive analysis harvests object properties from passing test cases and creates proper dynamic contracts.

properties from concrete states. In doing so, we note already collected properties in so-called *buckets* to align upcoming object events with them. Each bucket consists of dictionaries for observed method arguments, return values, and instance variables. Furthermore, each bucket includes mappings from the generalized object properties to the program entities where the concrete objects occur. With the help of harvesters and their buckets, we first derive common properties from concrete object events and compare them with already stored properties in the bucket. If a new object event includes a broader scope than stored in the bucket, we have to align the corresponding property. For example, the first concrete object is an integer with the value 1. So, we derive a value range for numbers from 1 to 1. If the second object event for the same program entity is a 10, we expand its value range from 1 to 10. After that a new object event with the number 5 would not influence the generalized object property. In our Figure 5.3, developers indicate interest in type and value range properties. Therefore, our inductive analysis tracer includes the two corresponding harvesters. While the first collects from concrete objects their most common super class type, the latter explores value ranges of primitive objects such as numbers, strings, and streams. In the type bucket, we find for the first argument of `initializeOn:` the common super class of all `Stream` types. In the range bucket, we store generalized stream properties such as information about its content and if it is already closed or not.

On the right, the inductive analysis is done and the collected object properties can be accessed by arbitrary tools. For our state navigation, we request this data in order to create dynamic contracts. We iterate over all generalized object properties and create contracts by calling a specific method at stored properties. This method returns a source code snippet that reflects the generalized object property as assertion. With the help of a contract builder [85], we aggregate all of these assertions into corresponding contracts for pre- and postconditions and invariants. For example, Figure 5.3 presents the created source code snippets for the `Stream` type and its value range. These assertions are then compiled as a contract and added to the corresponding `initializeOn:` method of the `WABufferedResponse` class.

Based on this architecture, we can implement arbitrary harvesters that collect generalized object properties for our dynamic contracts. Therefore, developers specify a new harvester that defines how common properties are derived from concrete objects and stored into the bucket. If this harvester is installed in our inductive tracer, object events automatically call corresponding interface methods for arguments, return values, and instance variables. Further on, each object property should implement a `printContract` method that converts its generalized data into source code assertions. Thus, we can automatically create dynamic contracts independent from the collected object properties. With the help of this framework, we have already implemented two specific harvesters which we describe in the following sections.

5.4.1. Type Harvester: Collecting Type Information

Our type harvesting gathers detailed type information of covered and executed program entities [103]. This is especially helpful in dynamically typed programming languages such as Smalltalk where type information is not explicitly represented in source code. For each method argument, return value, and instance variable that is executed during running tests, our type harvester receives the corresponding object event and derives its most common super class. We collect generalized type information by checking the type of the concrete object and comparing it to the deposited information in our type bucket. In the case that our bucket does not know the related program entity, we note the new type. If both types are equal or the new type inherits from the stored type, we do nothing because the stored information already comprises the new type. In all other cases, we store the common super class of both types. Therefore, we examine the class hierarchy and look for a class that both types inherit from[2]. Furthermore, our type harvesting also collects type information of container objects such as collections and dictionaries. Analogous to simple objects, we harvest each container element and summarize their types into a special container type. For the creation of dynamic contracts, we provide a generic implementation of the `printContract` method that creates the corresponding type assertions for all classes of the system.

In addition to our state navigation, we also reuse this type information for supporting

[2]In Smalltalk, there is a root class named `Object` that all objects have at least in common.

several other software engineering activities [103, 165]. With the help of additional type information, development tools can be improved by reducing result sets, checking for API conformance, and deducing concrete run-time behavior more precisely. This can help limit the scopes of search, navigation, and auto-completion tools to types actually used instead of all possible matching message signatures. Static analysis tools can check API usage to indicate related problems to give developers instant feedback about accidental mistakes while writing code. Also, program comprehension can be improved because developers know what types can be expected at dynamically typed variables.

5.4.2. Range Harvester: Checking Value Ranges of Objects

Value range harvesting collects common object properties for primitive data types such as numbers, collections, and strings. This information supports developers in understanding the value range of run-time objects, revealing violations in infection chains, and pointing out tests for missing corner cases. Table 5.1 summarizes all properties for primitive objects that we currently harvest. For example, in the case of number objects, we store its value range and check if it contains a zero value. If a specific object also includes the behavior of other primitive types, we check all of the corresponding properties, too. Since numbers inherit the behavior of objects, we also harvest whether number objects contain only constants and allow `nil` values.

To harvest all these common properties, we apply Smalltalk's libraries and meta-programming facilities. Having an object event from our harvester wrappers, we first check which behavior the corresponding object includes. With this data, we collect the common object properties with several libraries and reflection mechanisms. For example, the *spelling okay* property is verified by a spell checker that gets the string object as input. After that, we look up in our value range bucket whether other common object properties already exist. If not, we create a new range property with the first value ranges for this concrete object. Otherwise, we compare the new and existing value ranges and, if necessary, expand the common object properties. For example, we assume that `nil` objects are not allowed; however, as soon as an object event includes an undefined object we change this value to `true`. In this case, the object property denotes a final value that cannot be changed by subsequent events that do not include `nil` objects. In this way, we record object properties and generalize them step by step. Finally, to generate contracts, we represent each value range as a specific object that summarizes common properties for primitive types. Each range property implements the `printContract` method and knows how to convert its generalized data into proper assertions.

Behavior of	Property	Value range
Object	constant value	true/false
	nil allowed	true/false
Number	range	-∞ – ∞
	includes zero	true/false
Character	range	1 – 256
	includes letters	true/false
	includes digits	true/false
	includes separators	true/false
	includes specials	true/false
	is lowercase	true/false
	is uppercase	true/false
	is http safe	true/false
String	length	0 – ∞
	spelling okay	true/false
	includes numbers	true/false
	includes separators	true/false
	is ascii string	true/false
	content types	DateAndTime, Duration, Number, Time, FileDirectory
Collection	number of elements	0 – ∞
	fixed size	true/false
	empty allowed	true/false
Stream	contents	Set of classes
	is closed	true/false
DateAndTime	minimum date	0 – ∞
	maximum date	0 – ∞
	days of the week	1 – 7
	is leap year	true/false
	time zones	-11 – 12
Duration	seconds range	0 – ∞
	nano range	0 – ∞
	is positive	true/false
	is negative	true/false
	is zero	true/false

Table 5.1.: Harvested value ranges of primitive Smalltalk objects.

5.5. Summary

We presented our incremental dynamic analysis that provides the required run-time data for our test-driven fault navigation in a short time. We split the overhead of expensive dynamic analyses over multiple test runs and so ensure an experience of immediacy when debugging with our approach. This immediacy characteristic is realized by an interactive approach where developers incrementally decide about their needs. With these decisions, we automatically split the dynamic analysis over reproducible test runs and collect only the required information. Based on this idea, our incremental dynamic analysis consists of three specific techniques: Refined coverage analysis provides fast access to method coverage and on-demand refinements at statements for our structure navigation. Step-wise run-time analysis incrementally accesses the execution history of a specific test run for our behavior navigation. Inductive analysis harvests selected objects and generalizes their properties for our state navigation. All techniques have in common that they only collect initial data which developers can later incrementally refine on demand. As a result, we restrict not only the amount of data to the most necessary information, but also offer fast access to analysis results.

Part III.

Implementation and Evaluation

6 The Path Tools Framework

In this chapter, we outline the implementation of our Path Tools framework which realizes our test-driven fault navigation and incremental dynamic analysis. First, we start with an architectural overview of Path (Section 6.1), followed by a detailed description of our tool suite for the Squeak/Smalltalk development environment (Section 6.2). After that, we sketch the basic structure of our underlying incremental dynamic analysis framework (Section 6.3). Finally, we discuss our implementation with respect to other programming languages and its introduction costs to existing software systems (Section 6.4).

6.1. Architecture of Path

Test-driven fault navigation is based on our Path Tools framework which consists of 41 packages, 252 classes, 3,307 methods, and 17,559 lines of code. Figure 6.1 illustrates its architecture with respect to the underlying Squeak/Smalltalk[1] development environment [112].

At the top, our *Path tool suite* consists of the small helper tool PathProject; the extended test runner PathMap and its integrated developer ranking metric; the lightweight back-in-time debugger PathFinder; and the test-based source code editor PathBrowser. *PathProject* defines the scope of all further Path Tools and our incremental dynamic analysis. Therefore, this tool requires access to the source code in order to specify the system under observation. *PathMap* realizes both the structure and the state navigation of our approach. It needs Smalltalk's testing API to control the underlying unit test framework, the reflection API to determine a tree map of the system's structure, and the incremental dynamic analysis API to reveal anomalies with the refined coverage and inductive analysis. Our *developer ranking metric* is embedded into PathMap and combines its anomalies with author information of the source code change history. *PathFinder* allows developers to experiment with the execution history of specific test behavior and to follow infection chains back. It applies the testing and reflection API to control specific test runs and to show source code in corresponding call trees. Such call trees are built with the help of our incremental dynamic analysis API that starts step-wise run-time analysis and assigns anomalies. Finally, *PathBrowser* as our test-based source code editor provides access to entry points and invariants. It connects source code with already

[1] `http://www.squeak.org`

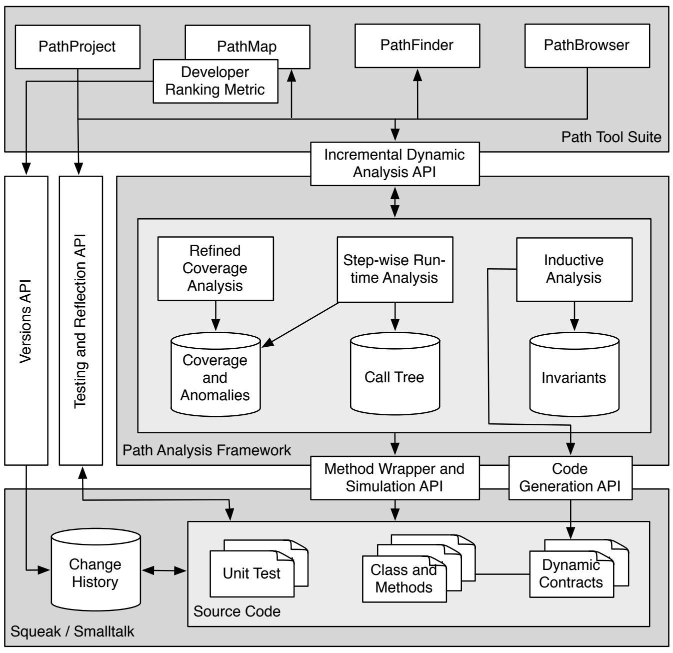


Figure 6.1.: The Path Tools framework is integrated into the Squeak/Smalltalk development environment and consists of a dynamic analysis framework and our tool suite.

recorded test coverage and generated assertions of our inductive analysis.

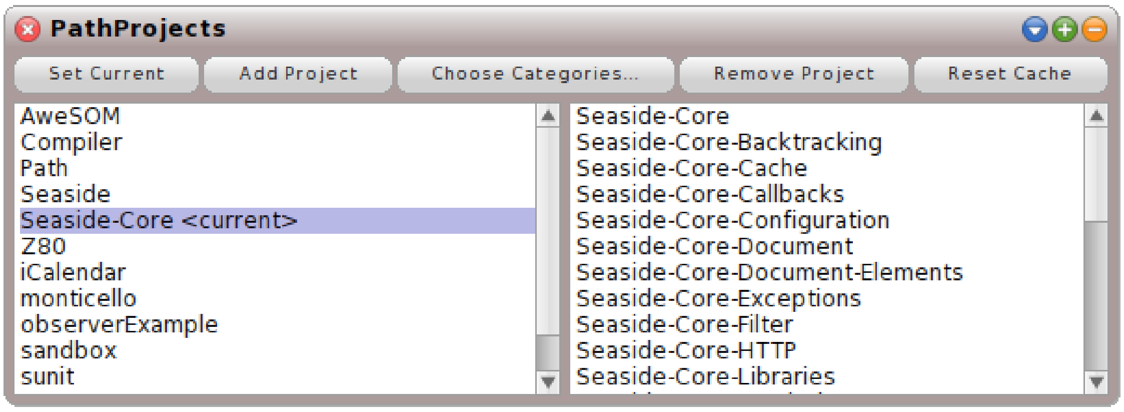
In the middle of our architecture, the *Path analysis framework* supports the observation of unit test behavior by implementing our incremental dynamic analysis. The *refined coverage analysis* rapidly records the relationship between unit tests and executed methods and refines statement coverage of selected methods on demand by re-executing their corresponding test cases. With the help of this analysis, PathMap reveals anomalies for our structural navigation within a short amount of time and at different levels of detail. *Step-wise run-time analysis* divides the dynamic analysis of one specific test case over multiple runs. PathFinder applies this analysis to record a simple method call tree, highlight anomalies in the execution history, and refine additional behavioral information. *Inductive analysis* harvests generic object properties and generates dynamic contracts of passing test cases for our state navigation. PathMap starts harvesting and creates contracts; PathFinder shows state anomalies if contracts are violated; and PathBrowser allows access to the generated assertions. We rely on method wrappers and Smalltalk's interpreter simulation for recording run-time information. At the level of methods, we collect run-time information with flexible method wrappers [40]: a wrapper introduces new behavior before and after the execution of a specific method without changing its original behavior. Depending on the chosen analysis technique, wrappers collect among others coverage, method calls, and state refinements. To record statements of a specific method, a special wrapper starts and stops Smalltalk's simulation engine that analyzes dedicated byte codes only. Both analysis techniques are necessary since a full simulation would slow down the execution by a factor of at least 100 [103]. Finally, the framework stores all collected measurements and makes this data available to any interested tool. For example, PathFinder can reuse comprehensive coverage information for highlighting anomalies in test case behavior.

At the bottom, we have implemented our Path Tools framework into the *Squeak/Smalltalk* development environment. Squeak is an open source implementation of the dynamic object-oriented programming language Smalltalk which only consists of objects and is image-based. This means that the complete system with all of its objects is persistently stored on disk. Thus, it is possible to enrich arbitrary objects with additional information such as their execution in test cases and to preserve this knowledge for later purposes. Squeak further offers a comprehensive development environment, a flexible and extendable user interface, and rich meta-programming facilities. As everything is written in Smalltalk, we have full access to all implementation details so that we can easily analyze, debug, and change the entire system. Squeak is used in a wide range of projects ranging from Web applications to several research prototypes. We also use this platform for teaching our students object-oriented programming by implementing small games and Web applications. For all these reasons, Squeak is an ideal basis for building our own debugging tools and to evaluate our approach with the help of our students and the community.

Our Path Tools framework only requires source code, unit tests, and a corresponding source code change history from the Squeak/Smalltalk system. We access and analyze these three program artifacts with our four different APIs. First, the *version API* allows access to the source code history that we require to determine experts for our developer ranking metric. In Squeak, all changes are automatically recorded by a built-in source code management system. After each change, author credentials, timestamp, and modifications are stored in a new version of the related program entity. Second, the *testing and reflection API* is required by all Path Tools in order to access unit tests and the structure of the system. It controls test executions, introspects program entities, and activates dynamic contracts. Third, the *method wrappers and simulation API* summarizes all meta-programming features for our incremental dynamic analysis. It offers a flexible way to implement arbitrary tracers and data structures that record run-time behavior such as call trees and invariants. Fourth, the *code generation API* is only required after harvesting invariants and allows for creating corresponding dynamic contracts in source code.

6.2. The Path Tool Suite

Our Path tool suite consists of four tools that together realize our test-driven fault navigation and offer worthwhile insights into program behavior. Before developers can start debugging with our approach, they have to define the system under observation and limit the scope of our incremental dynamic analysis. With PathProject, developers



AweSOM
Compiler
Path
Seaside
Seaside-Core <current>
Z80
iCalendar
monticello
observerExample
sandbox
sunit

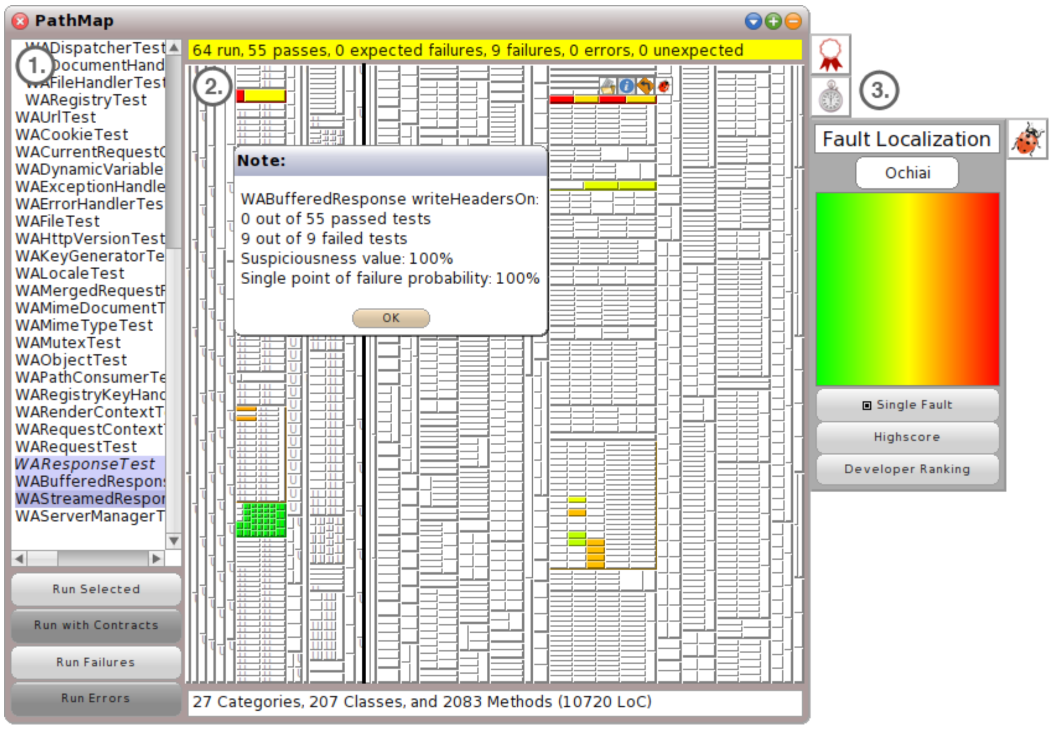
Seaside-Core
Seaside-Core-Backtracking
Seaside-Core-Cache
Seaside-Core-Callbacks
Seaside-Core-Configuration
Seaside-Core-Document
Seaside-Core-Document-Elements
Seaside-Core-Exceptions
Seaside-Core-Filter
Seaside-Core-HTTP
Seaside-Core-Libraries

Figure 6.2.: With PathProject, developers define the analysis scope of their system under observation with respect to packages and classes.

manage their different Smalltalk projects and describe which program entities are part of them. Having a project and a reproducible failure, all further Path Tools are ready to use. PathMap is our extended test runner that provides valuable feedback for restricting the initial search space. It realizes our structure navigation by computing spectrum-based anomalies and presenting the results in form of a tree map. Based on these results, it integrates our team navigation and its developer ranking metric. In addition to it, developers also start the inductive analysis for our state navigation from here, which harvests passing test cases in order to create contracts. Following our test-driven fault navigation process, PathFinder supports developers in debugging failing test cases back-in-time and highlighting the infection chain. It analyzes one specific test case and grants interactive access to its entire execution history. By mapping spectrum-based and state anomalies on it, we accomplish our behavior and state navigation. The last tool is not completely necessary for our test-driven fault navigation; however, it does connect the revealed test knowledge with source code and so supports program comprehension not only for debugging activities. PathBrowser offers test cases as entry points to arbitrary methods, enhances program entities with harvested run-time information, and shows the dynamically created contracts.

6.2.1. PathProject: Definition of the System under Observation

PathProject is a small helper tool in order to define the system under observation. Before developers can apply our Path Tools, they have to specify the project scope once by choosing their source code and testing packages of interest. As Squeak always allows access to the entire system with all its libraries, applications, and frameworks, we request developers to limit the analysis of our test-driven fault navigation to their needs. A project defines a partial trace that defines the system under observation and ignores remaining parts that are little likely to include failure causes. Nevertheless, if necessary the project



64 run, 55 passes, 0 expected failures, 9 failures, 0 errors, 0 unexpected

Note:

WABufferedResponse writeHeadersOn:
0 out of 55 passed tests
9 out of 9 failed tests
Suspiciousness value: 100%
Single point of failure probability: 100%

27 Categories, 207 Classes, and 2083 Methods (10720 LoC)

Figure 6.3.: PathMap is our extended test runner that analyzes test case behavior and visualizes suspicious methods of the system under observation.

scope can easily be extended by adding new source code packages. For example, in our typing error we limit the analysis to Seaside's core packages. In addition to partial traces, we use projects to cache specific meta data such as test coverage, anomalies, and harvested invariants. This data is specific to a project and is often required by multiple tools. So, we are able to interconnect our Path Tools and allow them access to already collected information. For example, PathFinder maps PathMap's spectrum-based anomalies into the execution history. Figure 6.2 presents our PathProject helper tool. Developers see all projects on the left and the corresponding packages of the selected project on the right. The buttons on the top serve to manage projects and to define the scope. The current project declares the active system under observation for our Path Tools.

6.2.2. PathMap: Extended Test Runner Feedback

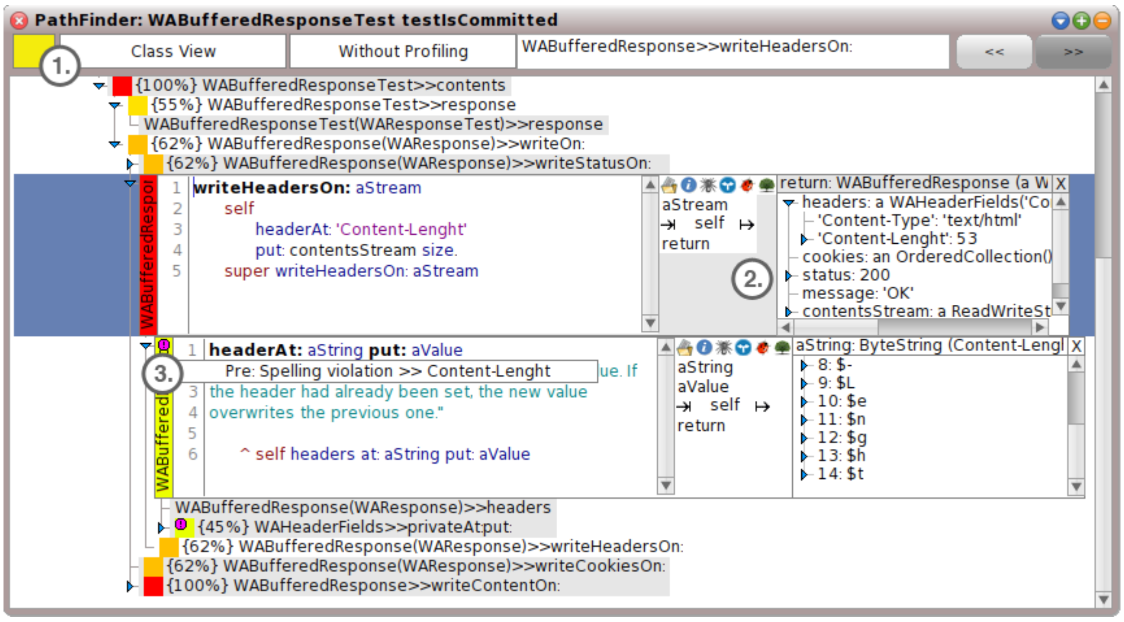
PathMap is an extended unit test runner for implementing our structure, team, and state navigation. Figure 6.3 shows our PathMap assigned to the Seaside Web framework. It does not only verify test cases but also localizes failure causes. Its integral components are from left to right a testing control panel, a compact tree map visualization of the software system, and several flaps for accessing various feedback techniques such as our structure navigation.

The testing control panel (1) provides nearly the same functionality as a standard test runner. Developers can choose from different test suites of the selected project and run them. Execution of test cases with enabled dynamic contracts can also be offered if assertions are available from the state navigation. In this case, we would highlight violations in our tree map analogous to the behavior navigation with small exclamation marks.

The compact visualization (2) provides valuable feedback about the project and its test cases. The tree map in the middle presents the structure of the system under observation with its packages, classes, and methods. Each small method box can be colored to represent test case and analysis results. It is possible for developers to interact with the tree map: hovering on a box results in the name of the attached method, its class, and its package; clicking on a box results in a menu being displayed. This menu allows developers to request additional information about the method such as its source code with refined statement coverage and the exact values of colored metrics (as shown in the message box). The menu also lets developers inspect the run-time behavior of the method by starting a covered test case entry point either with our PathFinder or a symbolic debugger. Furthermore, the test runner also presents a status bar on the top displaying a summary of the test suites execution and a status bar on the bottom displaying a summary of project metrics about the system.

Flaps on the right (3) set PathMap into specific analysis modes for collecting valuable feedback during the execution of test cases. Without an opened flap, PathMap acts like a standard test runner and only colors test case results in the tree map. In Figure 6.3, the fault localization flap is open and allows developers to start the structure navigation. If developers run selected tests, we automatically record their coverage, compute spectrum-based anomalies, and color the map with suspiciousness and confidence scores. Within the flap, developers can choose a proper spectrum-based metric such as Ochiai, see a legend explaining the colors in our tree map, and enable filtering of partially covered methods for single faults. Furthermore, the highscore lists all anomalies in descending order of failure cause probabilities and the developer ranking integrates our team navigation. After asking for a specific authorship metric, we collect all anomalies and present a ranked list of experts.

We offer several other flaps to further reveal hidden test knowledge and to control test runner feedback. The state navigation flap allows developers to harvest invariants from passing test cases. After choosing a specific harvester, our inductive analysis automatically collects generalized object properties and stores them in dynamic contracts later on. The test quality feedback flap [165] reveals the effectivity and efficiency of test cases. While the first relates coverage with static source code metrics for correcting low-quality tests, the latter finds common performance bottlenecks during the execution of test cases. The traceability flap connects test cases with arbitrary concerns and recovers the relationship to software artifacts. For example, we can link use cases with their acceptance tests and so reveal requirements traceability [105]. Moreover, we offer a global options flap that


PathFinder: WABufferedResponseTest testIsCommitted
Class View
Without Profiling
WABufferedResponse>>writeHeadersOn:
<<
>>
1.
2.
3.
WABufferedRespon
WABuffered

```
{100%} WABufferedResponseTest>>contents
  {55%} WABufferedResponseTest>>response
    WABufferedResponseTest(WAResponseTest)>>response
  {62%} WABufferedResponse(WAResponse)>>writeOn:
    {62%} WABufferedResponse(WAResponse)>>writeStatusOn:
```

```
writeHeadersOn: aStream
    self
        headerAt: 'Content-Lenght'
        put: contentsStream size.
    super writeHeadersOn: aStream
```

```
aStream
⇥ self ↦
return
```

```
return: WABufferedResponse (a W
headers: a WAHeaderFields('Co
  'Content-Type': 'text/html'
  'Content-Lenght': 53
cookies: an OrderedCollection()
status: 200
message: 'OK'
contentsStream: a ReadWriteSt
```

```
headerAt: aString put: aValue
    Pre: Spelling violation >> Content-Lenght    ue. If
    the header had already been set, the new value
    overwrites the previous one."

        ^ self headers at: aString put: aValue
```

```
aString
aValue
⇥ self ↦
return
```

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

```
      WABufferedResponse(WAResponse)>>headers
      {45%} WAHeaderFields>>privateAt:put:
      {62%} WABufferedResponse(WAResponse)>>writeHeadersOn:
    {62%} WABufferedResponse(WAResponse)>>writeCookiesOn:
    {100%} WABufferedResponse>>writeContentOn:
```

Figure 6.4.: PathFinder is a lightweight back-in-time debugger that classifies failing test behavior for supporting developers in navigating to failure causes.

allows developers to filter primitive methods, ignore test setups, visualize intermediate steps, show test results, and optimize harvesting.

6.2.3. PathFinder: Lightweight Back-in-time Debugger for Test Cases

PathFinder is our lightweight back-in-time debugger for introspecting specific test case executions with a special focus on fault localization. Not only does it provide immediate access to run-time information, but also classifies traces with suspicious behavior and state. To localize failure causes in behavior, developers start exploration either directly at a failing test case or out of covered suspicious methods as provided by PathMap. Subsequently, PathFinder opens at the chosen method as shown in Figure 6.4 and allows for following the infection chain back to the failure cause. Its main components are a control panel on the top and the test case call tree representing its execution history.

At the first index in Figure 6.4, PathFinder provides a control panel to set up the dynamic analysis of test cases and to support the navigation through the large amount of run-time data. From left to right, it offers the following functionality. The yellow box presents the final test result and allows developers to explicitly choose and rerun test cases. Views and profiling information influence the shallow analysis of our step-wise run-time analysis and enhance the results of initial call trees. While views record more details about called receiver objects such as names and identities, profiling precisely measures the required time for executing a test case. Although both techniques are useful with respect to follow a particular object or to identify performance bottlenecks, their analyses require more time. For that reason, developers can optionally refine the call tree with this additional data. A small query engine and history buttons complete the control panel by providing functionality for navigating through an execution trace. So far, the query engine allows developers to search for called methods, anomalies, and additional information such as views and profiling information.

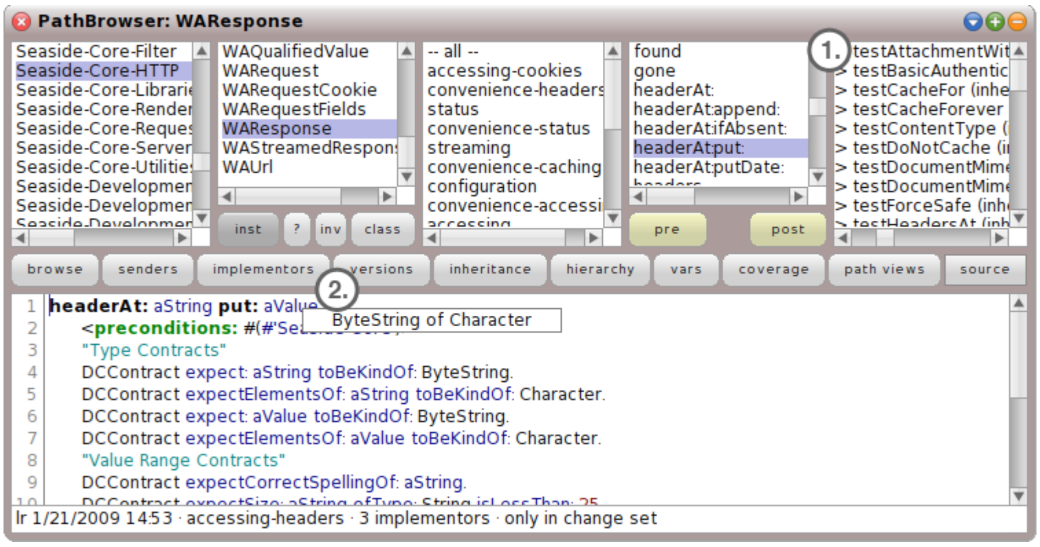
At the bottom of Figure 6.4, the visualized information primarily consists of a call tree that reflects a particular test case run. A call tree provides comprehensive information of the entire program execution and shows how methods call each other to fulfill the test case behavior. From top to bottom, each node represents one method call and their subtrees describe its called methods. Each method call node consists of a colored box with a percentage value for its suspiciousness score and a name representing receiver class, implementation class in parentheses, and method name. Optionally, this name also includes view and profiling information. We provide arbitrary navigation through method call trees and their state spaces. Developers can follow traces in both directions and expand and collapse subtrees interactively. For a better distinction, we alternate the background colors of called methods depending on their stack depth.

Some of the tree nodes have been expanded to reveal details about the method implementation and the applied state. At the second index in Figure 6.4, an expanded method call node shows its source code, a control panel for requesting details and refining run-time data, and the return value. Most notably, the control panel allows developers to start a source code editor and a symbolic debugger, obtain additional information about anomalies, refine coverage and spectrum-based fault navigation at statements, and explore object states. After indicating interest in a specific argument, receiver, or return object, we reexecute the test case, make a deep copy of the requested object, and present it in an object explorer on the right. Developers can explore all object properties and compare them to other method nodes in the execution history.

The third index in Figure 6.4 refers to our highlighted state navigation. We map violated contracts to traces by adding small purple exclamation marks to method nodes. We choose this color because of its availability and good visibility. Developers can further inspect such exclamation marks and receive detailed information about the exact violation. For example, the shown label presents the spelling violation of our Seaside typing error.

6.2.4. PathBrowser: Connection of Source Code and Hidden Test Knowledge

PathBrowser extends Squeak's standard source code browser with hidden test knowledge to improve further program comprehension. Figure 6.5 shows our two extensions. First, we offer test cases as entry points into behavioral examples of specific methods. We reuse the already collected test coverage data from PathMap and present it in the new pane on the right side. With a click on a corresponding test case, developers can start a symbolic



```
1  headerAt: aString put: aValue
2      <preconditions: #(#'Se
3      "Type Contracts"
4      DCContract expect: aString toBeKindOf: ByteString.
5      DCContract expectElementsOf: aString toBeKindOf: Character.
6      DCContract expect: aValue toBeKindOf: ByteString.
7      DCContract expectElementsOf: aValue toBeKindOf: Character.
8      "Value Range Contracts"
9      DCContract expectCorrectSpellingOf: aString.
```

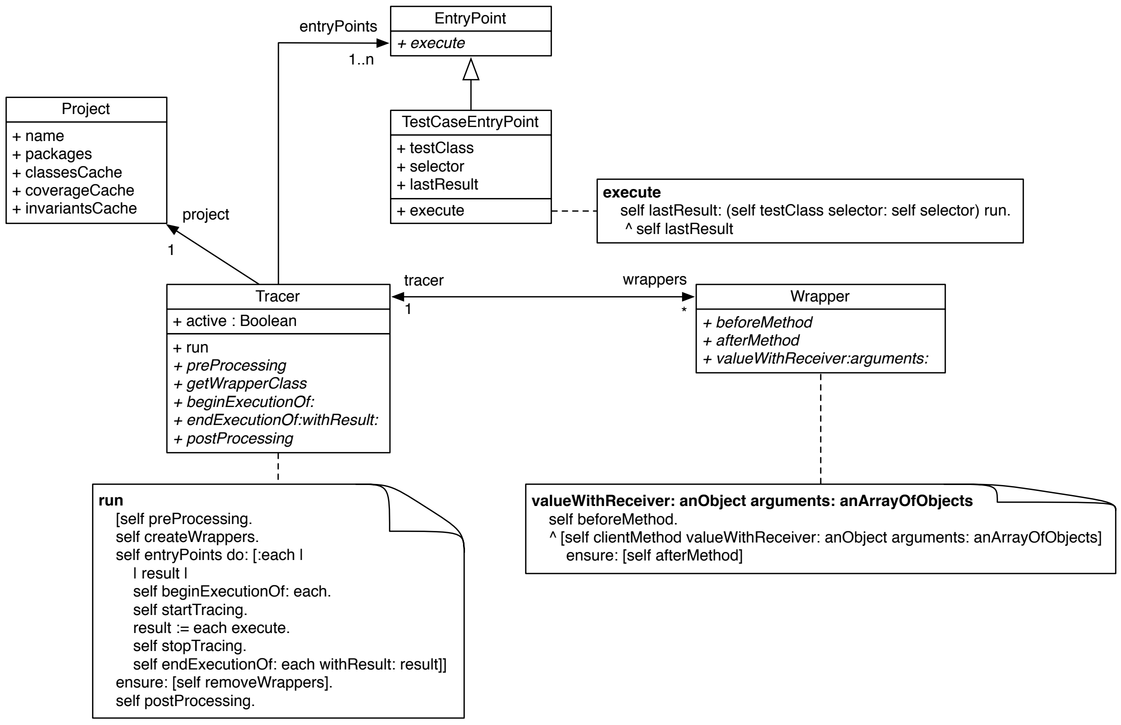
```
lr 1/21/2009 14:53 · accessing-headers · 3 implementors · only in change set
```

Figure 6.5.: PathBrowser connects the hidden test knowledge with source code.

debugger or our PathFinder. Both tools stop their execution at the first call of the selected method and provide insights into its concrete run-time behavior. Thus, developers learn how a method works and so comprehend source code abstractions by debugging into examples. Second, we integrate the hidden test knowledge into source code by presenting invariants and dynamic contracts of our state navigation. If a specific program entity possesses generalized object properties, then a label automatically shows its exploited run-time information while editing source code. For example, we harvested the `ByteString` class as type for the argument `aValue`. This information allows developers to understand source code better, especially in dynamically typed languages such as Smalltalk where type information is rather implicit. Furthermore, we extend the browser with our dynamic contracts. There are buttons for displaying the source code of invariants (inv), pre- and post-conditions (pre/post). The Smalltalk code at the bottom shows all the generated assertions from our collected invariants. Developers can also add manual assertions to this source code. In the example of Figure 6.5, we present the pre-condition contract for the method `WAResponse»headerAt:put:`. The assertion `DCContract expectCorrectSpellingOf: aString` (line 9) throws a violation in our Seaside typing error and reveals the crucial state anomaly.

6.3. The Path Analysis Framework

Path also provides a flexible and extendable framework for implementing our incremental dynamic analysis techniques. We offer a generic infrastructure with several hooks that supports the experimentation of new dynamic analysis techniques. Figure 6.6 sketches a simplified version of Path's core classes and template methods [85].



```
execute
    self lastResult: (self testClass selector: self selector) run.
    ^ self lastResult
```

```
run
    [self preProcessing.
    self createWrappers.
    self entryPoints do: [:each |
        | result |
        self beginExecutionOf: each.
        self startTracing.
        result := each execute.
        self stopTracing.
        self endExecutionOf: each withResult: result]]
    ensure: [self removeWrappers].
    self postProcessing.
```

```
valueWithReceiver: anObject arguments: anArrayOfObjects
    self beforeMethod.
    ^ [self clientMethod valueWithReceiver: anObject arguments: anArrayOfObjects]
        ensure: [self afterMethod]
```

Figure 6.6.: The simplified core of our Path analysis framework consists of projects, entry points, tracers, and wrappers.

`Project` is a helper and data class that represents the partial trace of the system under observation. After developers have defined the analysis scope with PathProject, a corresponding project object represents its name, packages, and related classes. Each tracer applies the current project to not only read its information but also to finally cache the analyzed results. Hence, all Path Tools have access to gathered run-time data and can reuse it.

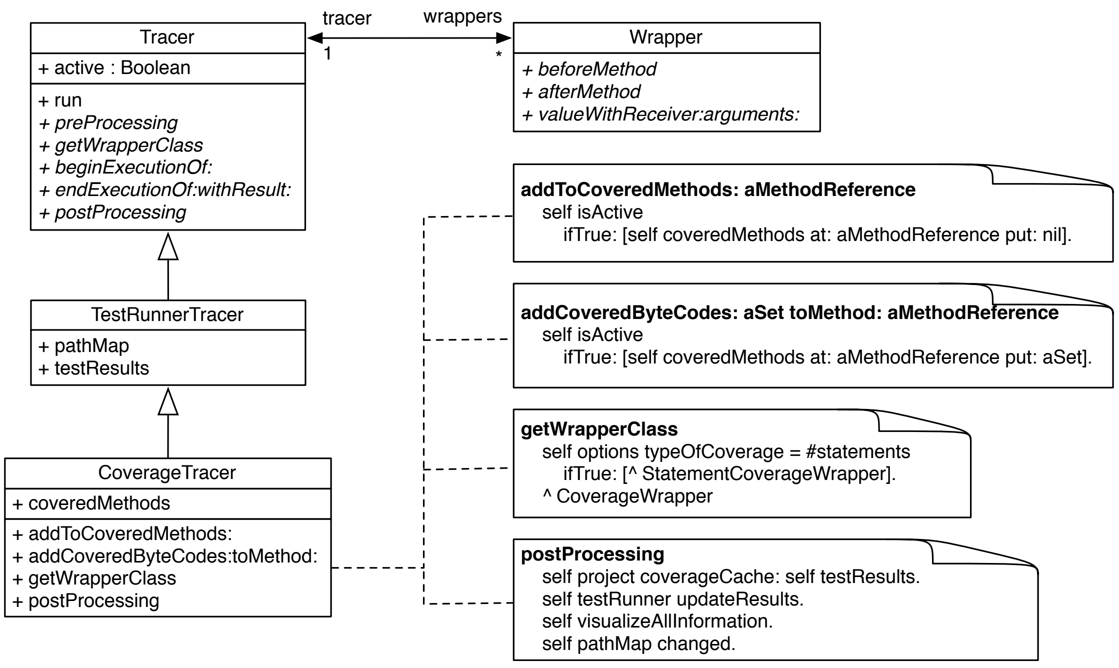
`EntryPoint` represents our concept of starting points into reproducible and deterministic behavioral paths through the system. It is an abstract class because we offer several kinds of reproducible entry points such as test cases, source code scripts, and business-readable acceptance tests [45]. Since our test-driven fault navigation requires test cases as entry points, we focus on the corresponding subclass. `TestCaseEntryPoint` is an adapter [85] that encapsulates a test case with its class and selector. Furthermore, it stores the last test result to promptly compute anomalies without the need of running test cases again. Finally, the `execute` method implements the `EntryPoint` hook by executing the included test case and caching its result.

`Tracer` is the centerpiece of our framework as it controls the dynamic analysis process. It is a generic and adaptable coordinator that defines which system parts are observed and what information is collected. Based on the project scope, a specific tracer creates and removes method wrappers, executes assigned entry points, and aggregates the run-time data being collected by its method wrappers. The `run` method summarizes this process and provides several template methods [85] that can be implemented by subclasses to fulfill their specific dynamic analysis needs. We start with a preprocessing that can adapt the system under observation or initialize data structures such as call trees and harvesting buckets. After that, we create and install wrappers all over the system under observation. In doing so, subclasses define the specific wrapper class and can further adapt the analysis scope. Having an instrumented system, we execute each entry point for its own. In the course of this, we offer hooks before and after each execution and for the activation of tracing. For example, subclasses can declare interest in partial results and delay the activation to ignore the analysis of setup code. After running all entry points, we have to ensure that all wrappers are removed from the system in order to prevent inconsistencies. Finally, a last postprocessing step allows subclasses to finally aggregate and present collected run-time values.

We analyze the execution of test cases by instrumenting the code using method wrappers [40]. `Wrapper` is a very lightweight construct that transparently replace methods with alternative implementations and can delegate to the original (wrapped) functionality. As long as a wrapper is dynamically installed, its `valueWithReceiver:arguments:` method is called instead of the original method. With this hook, subclasses can collect specific run-time data before and after they forward the call to the client method. In doing so, each wrapper has access to its tracer object in order to send collected events. The tracer summarizes all wrapper events and aggregates the run-time data together.

As an example of our framework, we sketch the concrete implementation of our refined coverage analysis that allows fast access to method coverage and on-demand refinements at statements. Figure 6.7 illustrates the specific tracers and their implemented hooks. The `TestRunnerTracer` inherits the basic functionality of `tracer`, connects it to our PathMap tool, and provides a generic data structure for test results. This tracer only executes test cases, stores their results, and visualizes them in PathMap's tree map. It is a standard test runner implementation that executes test cases and renders their results without any further analysis. In addition to it, the class offers a generic data structure for all following incremental dynamic analysis tracers called `testResults`. With the help of a dictionary, the tracer puts test cases and their results as keys and further analysis data as values. The `CoverageTracer` implements the refined coverage analysis. For each test case run, it stores covered methods with optional statement refinements in the `coveredMethods` dictionary. While keys reflect covered methods, values can describe covered statements. At the end of each test case run, this dictionary is stored as value in the generic `testResult` data structure. To fill the `coveredMethods` dictionary, we offer two *add* methods being called by executed method wrappers. As long as the tracer is active, `addToCoveredMethods:` simply stores covered method references and `addCoveredByteCodes:toMethod:` further adds covered byte codes as additional data[2]. Our tracer offers two kinds of wrappers to record coverage

[2]The storage of byte codes allows for deriving covered statements later on. Furthermore, the simplified `addCoveredByteCodes:toMethod:` implementation hides the merge of previously covered byte codes.



```
addToCoveredMethods: aMethodReference
    self isActive
        ifTrue: [self coveredMethods at: aMethodReference put: nil].
```

```
addCoveredByteCodes: aSet toMethod: aMethodReference
    self isActive
        ifTrue: [self coveredMethods at: aMethodReference put: aSet].
```

```
getWrapperClass
    self options typeOfCoverage = #statements
        ifTrue: [^ StatementCoverageWrapper].
    ^ CoverageWrapper
```

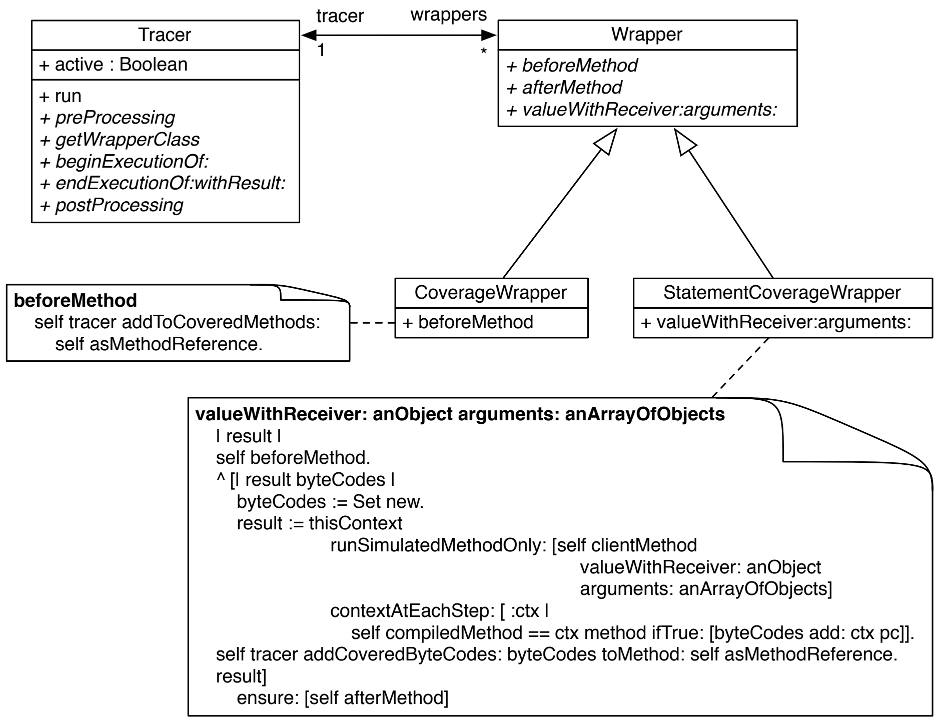
```
postProcessing
    self project coverageCache: self testResults.
    self testRunner updateResults.
    self visualizeAllInformation.
    self pathMap changed.
```

Figure 6.7.: Path's specific tracers for our refined coverage analysis.

at methods and statements. Depending on developers' needs, the method `getWrapperClass` returns the corresponding wrapper type. Furthermore, in the case of the more specialized `StatementCoverageWrapper` we also limit the instrumentation to the chosen method only. At the end, the postprocessing stores the final results into the coverageCache and triggers the visualization of the collected information in PathMap.

Figure 6.8 shows the related two wrappers for our refinement analysis. `CoverageWrapper` only has to implement the `beforeMethod` that is always executed before the original method. It sends a message event to its corresponding tracer that the wrapped method is covered. The `StatementCoverageWrapper` is more complicated because the refinement of covered statements requires Smalltalk's byte code simulation. Such a bytecode simulator executes code in a simulated mode that allows additional functionality to intercept and analyze execution at each interpreter step. However, this kind of dynamic analysis is expensive and should be limited to important methods only. For that reason, we only analyze methods that developers request. In our `StatementCoverageWrapper`, the method `valueWithReceiver:arguments:` is overwritten and calls the original method in simulation mode. At each new context, we store the executed byte code with respect to the program counter. After the simulation, we forward the recorded byte code to our tracer and return the result of the original method.

To implement our incremental dynamic analysis completely, we provide several specific subclasses of `Tracer` and `Wrapper`. Table 6.1 gives an overview for each analysis. Our



```
beforeMethod
  self tracer addToCoveredMethods:
     self asMethodReference.
```

```
valueWithReceiver: anObject arguments: anArrayOfObjects
   | result |
   self beforeMethod.
   ^ [| result byteCodes |
      byteCodes := Set new.
      result := thisContext
                  runSimulatedMethodOnly: [self clientMethod
                                              valueWithReceiver: anObject
                                              arguments: anArrayOfObjects]
                  contextAtEachStep: [ :ctx |
                     self compiledMethod == ctx method ifTrue: [byteCodes add: ctx pc]].
   self tracer addCoveredByteCodes: byteCodes toMethod: self asMethodReference.
   result]
      ensure: [self afterMethod]
```

Figure 6.8.: Path's specific wrappers for our refined coverage analysis.

refined coverage analysis requires a coverage tracer for collecting the pure relationship between tests and methods and a more specialized fault localization tracer for additionally computing anomalies. It also requires two wrappers to collect coverage at different levels of detail. Our step-wise run-time analysis has one tracer that builds and refines the call tree step by step. For that purpose, it relies on different kinds of method wrappers to decorate methods with additional functionality required during our shallow and refinement analysis. While the call wrapper and its subclasses view and profiling wrapper collect data for constructing the call tree, all other wrappers are specific to one method for debugging it or exploring deep copies of objects. Our inductive analysis encapsulates harvester buckets and aggregates object states from harvester wrappers. Furthermore, all tracers have a common super class named `TestRunnerTracer`. This class connects tracers with PathMap and provides basic functionality to execute test cases, record their results, and visualize them in our tree map. In summary, we have implemented all incremental dynamic analysis techniques with the same framework, which demonstrates its flexibility and extensibility. We argue that our framework is also able to implement other dynamic analyses such as efficiency of test case [165] and requirements traceability [105].

Incremental dynamic analysis	Tracer	Wrapper
Refined coverage analysis	`CoverageTracer`	`CoverageWrapper`
	`FaultLocalizationTracer`	`StatementCoverageWrapper`
Step-wise run-time analysis	`CalltreeTracer`	`CallWrapper`
		`ViewWrapper`
		`ProfilingWrapper`
		`SpecificWrapper`
		`DebugWrapper`
		`ExploreWrapper`
Inductive analysis	`InductiveAnalysisTracer`	`HarvesterWrapper`

Table 6.1.: Specific subclasses of tracer and wrapper for implementing our incremental dynamic analysis.

6.4. Discussion

We argue that our approach can easily be adapted to other object-oriented programming languages that include a unit test framework. To implement our Path Tools framework, the language and its libraries have to support dynamic and static analysis techniques. While the dynamic analysis of method executions can be implemented with aspect-oriented programming [91], statement-level coverage depends on language features. For example, in C++ many coverage tools insert probes into the source code. In Python, the interpreter offers a simple hook function for a fine-grained run-time analysis. Regarding static analysis, developers can rely on several external analysis tools or the reflection capabilities of their language. Also, many version control systems such as subversion offer interfaces to request author information of previous changes. Finally, our Path tool suite to large parts is a visualization concept whose implementation only depends on the underlying user interface of the development environment. For example, Eclipse could also be extended with a plug-in for rendering the tree map and its anomalies.

The introduction costs for test-driven fault navigation are low in the beginning but it will take some time to become well acquainted with our Path Tools. These costs are largely composed of adapting the underlying software system, teaching our new debugging process, and practicing with our tools. The preparation effort for the underlying software system is negligible. Once our Path Tools framework is available for a specific programming language, it only requires access to source code, unit tests, and the change history. We have already analyzed numerous Smalltalk projects without any problems. Only in very few instances, did we have to revise non-deterministic unit tests or we could not analyze the system because of extensive reflection mechanisms. The new concepts of our test-driven fault navigation process are easy to understand because they are similar to other debugging activities. In comparison to the *traffic* principle [219], developers also start with a breadth-first search and they systematically follow the infection chain back. For example, after presenting our approach in a 90-minutes lecture, our undergraduate students confirmed that they have understood our approach. They praised the possibilities to refine hypotheses with anomalies and to navigate the infection chain backwards. However, we also learned from our students that practice with our Path tool suite needs some time. Although they become acquainted with the main features within a few hours, an efficient debugging session still looks different. On the one hand, highlighted anomalies in PathMap and PathFinder as well as the developer ranking are easily applicable. On the other hand, debugging back in time and searching anomalies in large traces require a difficult change of thinking when localizing more complicated failure causes. For these reasons, we will prepare additional tutorials for getting started with the more advanced features of our tools.

6.5. Summary

We introduced the implementation of our Path Tools framework for the Smalltalk programming language and Squeak development environment. Path realizes both our test-driven fault navigation for debugging reproducible failures and our incremental dynamic analysis to ensure an immediate experience when applying our tools. We started with a compact overview of its architecture presenting the relationships and APIs between Path Tools, our analysis framework, and the underlying Squeak/Smalltalk system. Based on this architecture, we explained each Path tool in more detail. PathProject is a little helper tool to define the analysis scope and the system under observation. PathMap is our extended test runner for starting debugging with our test-driven fault navigation. PathFinder is our lightweight back-in-time debugger for following failing test cases back to their root causes. PathBrowser connects the hidden test knowledge with source code and so enhances several program comprehension tasks. After the description of our tools, we sketched the analysis framework by introducing its core classes and their hooks. Tracers in combination with method wrappers allow for implementing arbitrary dynamic analysis techniques. We showed the flexibility and extensibility of this mechanism by explaining the refined coverage in more detail and presenting the concrete subclasses for each incremental dynamic analysis technique briefly. At the end, we discussed our approach with respect to porting it to other programming languages and its introduction costs into existing projects.

7 Studies and Experiments

In this chapter, we evaluate our test-driven fault navigation and incremental dynamic analysis with respect to its practicality, effectiveness, and efficiency. With the help of our Path Tools framework, we examine several Smalltalk projects (Section 7.1) and numerous failures in order to prove that our approach is able to reduce time and effort required for debugging. We start with a user study (Section 7.2) that observes and compares developers during debugging with a symbolic debugger and our Path Tools. After that, we consider the quality of our automatic test-driven heuristics, in particular our developer ranking metric (Section 7.3) and inductive analysis (Section 7.4). Finally, we measure the performance characteristics of our implementation (Section 7.5). In summary, we learn that our novel debugging approach and its corresponding tools reduce not only the required debugging time but also assist developers in efficiently creating, evaluating, and refining failure cause hypotheses.

7.1. Projects Studied

To evaluate our approach, we applied test-driven fault navigation and its corresponding Path Tools framework to six Smalltalk projects. The properties of the mid-sized projects and their specific evaluation purposes are summarized in Table 7.1. Of the six projects, two (AweSOM and zEmu) are research prototypes developed in our group, one project (4Conferences) is a long-lived student project, and the remaining three are external, production-quality projects. All projects are well tested, not implemented by the author of this dissertation, and in daily use in software development and business activities. Their acceptance, integration, and unit tests cover large parts of the system, imposing different computational cost. These projects involve various application domains such as end-user Web frameworks, development tools, and virtual machines. As a consequence, we argue that our Path Tools framework is applicable to a broad range of different software systems.

Seaside [64, 169] is an open source and industrial Web framework[1] that includes our motivating typing error. In its version 3.0, Seaside allows the easy creation of powerful Web applications using high-level abstractions on the application components and on the underlying hypertext transfer protocol. In doing so, it builds upon the strengths of the

[1] `http://www.seaside.st`

	Seaside	iCalendar	4Conferences
Classes	657	77	175
Methods	5564	1347	2540
Tests	711	115	89
Coverage	41.4 %	72.9 %	69.3 %
Evaluation	Motivating example Efficiency	User study Efficiency	Team navigation Efficiency
Section	2.1, 7.5	7.2, 7.5	7.3, 7.5

	AweSOM	Compiler	zEmu
Classes	68	64	47
Methods	750	1294	1012
Tests	125	49	349
Coverage	81.8 %	51.1 %	91.7 %
Evaluation	State navigation Efficiency	Efficiency	Efficiency
Section	7.4, 7.5	7.5	7.5

Table 7.1.: Project characteristics for our user, effectiveness, and efficiency studies.

Smalltalk object-oriented programming language and transcends many of the common practices needed in other, less dynamic languages.

We choose Squeak's *iCalendar* project[2] as the underlying software system for our user study. iCalendar is a library that supports the identically named file format for sharing meeting requests and tasks independent of specific calendar applications. The project implements import and export functionality of the file format including a parser, a domain-specific object model, and I/O handling. It is an external, open source, and real-world project that is used in several other applications. We choose iCalendar because of its maturity, already included and comprehensive test base, understandable domain, and ideal project size that is neither too small nor too large.

4Conferences supports the web-based management of conferences and determines the accuracy of our developer ranking metric. The conference management system permits activities such as the registration of attendees, the organization of payments, the printing of badges, and the planning of talks. It has been developed by five undergraduate student projects in the context of two software engineering courses in the last two years. In these courses, we acted as customers for the 28 involved students. The students implemented 4Conferences with the help of Extreme Programming. Due to agile practices such as collective code ownership, all developers had access to the entire code base and each

[2] `http://www.squeaksource.com/ical/`

method was implemented by a number of students. Thus, this project has a sufficient distribution of developers for evaluating our team navigation.

AweSOM [102] is a research prototype developed in our group that realizes a virtual machine for running and interpreting SOM's file-based Smalltalk dialect on top of the Squeak system. We evaluate the quality of our state navigation in the context of this project as it is very well suited for such a task for two reasons. First, there are two independent test suites provided by different authors that can serve as measured and reference sets for our evaluation. The acceptance test suite consists of a collection of SOM Smalltalk applications that are executed by the virtual machine; they are real-world examples whose execution involves the entire virtual machine. The unit and integration test suite verifies the internal virtual machine components and their collaboration. Second, the domain of a virtual machine implementation brings about several different object states that can exhaust our harvesters.

The remaining two (*Compiler* and *zEmu*) are additional projects that have also profited from our Path Tools framework during their maintenance. While the first project refers to Squeak's standard Smalltalk compiler, the second is an emulator prototype for an experimental programming language from our group. We consider both projects only for performance characteristics.

Apart from these six projects, we have used our Path Tools framework in a number of other projects not covered in this evaluation. For several terms, students have applied our tools in our software engineering courses and bachelor projects. With the help of our approach, they gained valuable feedback about their implemented games, Web applications, and research prototypes. For example, the Orca Web application framework [199], which translates Smalltalk code to JavaScript, solved many difficult failures with our test-driven fault navigation. In almost all projects, results showed that our Path Tools could be used in place of the standard test runner and debugger without any adaptions to the underlying system.

7.2. User Study: Practicality of Test-driven Fault Navigation

To evaluate test-driven fault navigation and its corresponding tools, we conduct a user study within the scope of the *iCalendar* project (see Section 7.1). We observe six developers while debugging six failures and compare PathMap and PathFinder with Smalltalk's test runner and symbolic debugger. In summary, we discover that test-driven fault navigation is able to decrease debugging cost with respect to required time and developer's effort.

Failure	Description	Difficulty	Infection chain length	Suspicious method rank
1	Reversed comparison operator of event objects	Easy	1	1
2	Wrong conditional statement in handling address parameters	Easy	27	22
3	Unintended string constant in phone types	Normal	5	68
4	Forgotten deletion of obsolete calendar events	Normal	84	10
5	Missing separator for parsing event files	Hard	520	31
6	Return of improper but polymorph objects for storing alarms	Hard	2133	42

Table 7.2.: Description of iCalendar's failures.

7.2.1. Experimental Setup

For our user study, we observe six developers with a similar background knowledge. The participants are computer science students in the 5th semester. All of them have between 3 and 11 years of programming experience and professional expertise with symbolic debuggers. In the last year, they attended two of our software engineering courses, in which we intensively taught object-oriented programming with the help of Smalltalk. In this time, they built a new system from scratch, maintained an existing application, and passed the courses with excellent grades. The chosen students have similar development skills and become acquainted with iCalendar in our user study for the first time. Thus, we can ensure that the required debugging effort is not much influenced by individual skills and knowledge about the system.

During the study, these students are supposed to localize six failures with different levels of difficulty, which are described in Table 7.2. We insert these six defects all over the iCalendar system, whereby we described them obviously. For example, we comment important statements instead of deleting them. So, it is easier for our developers to verify defects after they have followed the infection chain backwards. For each failure, we assess a level of difficulty that estimates the required effort to localize its defect. On the one hand, we choose this level depending on our own debugging experience; on the other hand, it also reflects the number of call nodes between failure and defect (infection chain length) and the position of the defect in the list of anomalies (suspicious method rank). Finally, we have two failures from each of the six, which are easy, normal, and hard to debug. All failures can be reproduced with 1-10 failing test cases, which do not trivially include defects as part of their stack traces.

7.2.2. User Study Procedure

Prior to our user study, we presented a software engineering course in which all students had already access to our Path Tools for a period of three months. In this course, we also taught them advanced debugging concepts [219] and we introduced test-driven fault navigation as an exemplary approach. To find participants for our user study, we performed a short questionnaire with all students, which takes into account years of programming experience, reviews of course projects, and grades. Based on these results, we selected six excellent students with a similar background knowledge. However, the questionnaire also revealed that the students' main preoccupation was effecting test-driven fault navigation because of the new debugging concept and neglected guidance from our part.

For the preparation of our participants, we introduced test-driven fault navigation again and we allowed them to become acquainted with iCalendar. Within two hours, we first presented our Seaside typing error followed by an instructed and comprehensive practice with our tools. The students debugged one example failure in iCalendar under our guidance. In doing so, they understood iCalendar's basic concepts, investigated its source code, and learned to debug with test-driven fault navigation.

We conducted the user study by observing our developers while debugging iCalendar's failures with different tools. First, all developers debugged three failures with Squeak's standard debugging tools. After that, they debugged the remaining three failures with our Path Tools. While using test-driven fault navigation, we provided assistance with handling PathFinder's user interface and its features. The process of localizing failure remained free of influence. For all six failures, we measured the complete debugging time, including everything from time to consider source code to execution time of applied tools. Additionally, for test-driven fault navigation we noticed the point in time when our developers started PathFinder to follow the infection chain back. If the defect was not localized after 15 minutes, we marked the failure as not solved.

To evaluate our approach, we assigned each developer three failures for debugging with standard tools (symbolic debugger and test runner) and three failures for test-driven fault navigation (PathFinder and PathMap). For each of the six failures, we ensured a unique combination of developers and applied tools. At each level of difficulty, a developer debugged one failure with standard tools and the other one with our Path Tools.

After the study, we interviewed each student and asked for feedback with respect to our approach and Path Tools.

7.2.3. Discussion of Study Results

With the help of our user study, we reveal that test-driven fault navigation is able to reduce the required effort for localizing failure causes. Compared to standard debugging

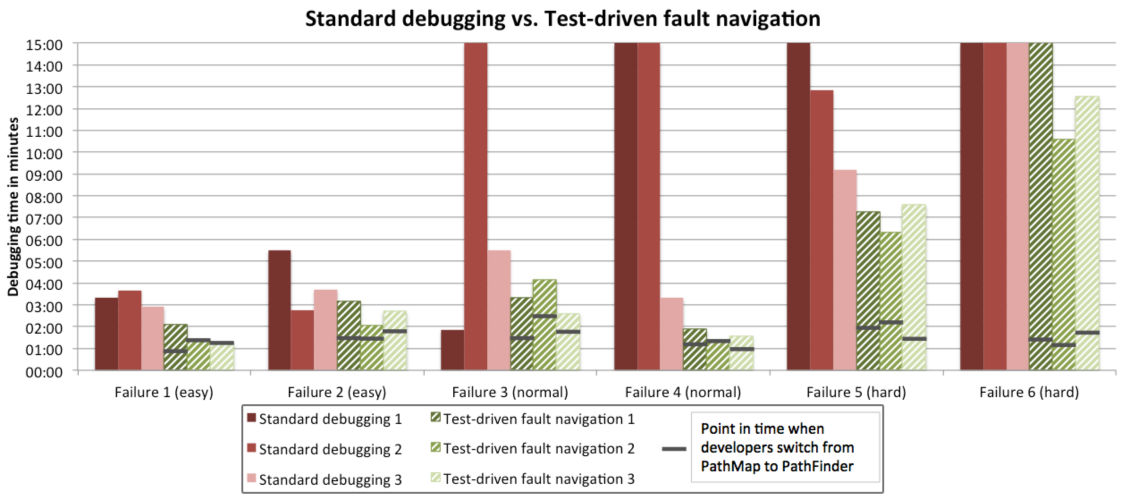


Figure 7.1.: Required debugging time with symbolic debugger and test runner compared to PathFinder and PathMap.

tools, developers who apply our Path Tools need in the majority of cases less time for debugging. Figure 7.1 summarizes the required debugging time for each failure with standard tools and our Path Tools. The time includes all applied debugging activities such as running tests, time to consider source code, and execution time of tools. For each level of difficulty, the position of a bar corresponds to the same developer with respect to the applied tools per failure. For example, the second bar in normal failure 3 standard debugging and the second bar in normal failure 4 test-driven fault navigation represent the same developer.

In the case of the first two easy failures, developers with test-driven fault navigation are about one minute faster compared to developers with symbolic debuggers and test runners only. For example, while failure 1 requires at least three minutes to debug with standard tools, our Path Tools are able to localize the defect in less than two minutes. Even if all developers exchange their tools for failure 2—from standard tools to Path Tools and vice versa—test-driven fault navigation is almost always faster. Thus, our performance increase is independent of the expert knowledge of individual developers.

We have similar results with the next two normal failures, but the differences in required debugging time are considerably larger. In particular, developers with standard tools have some problems in localizing failure causes. Three students could not find the defect within 15 minutes because they did not comprehend what is going wrong. In contrast, two other developers required only a short time for debugging with standard tools. While one developer knows the infected source code very well from the preparation phase, the other developer instantly had a proper intuition. Nevertheless, test-driven fault navigation is once more better and allows all developers to localize failure 3 in less than four minutes and failure 4 in less than two minutes. Compared to standard debugging tools, the integration of anomalies into a systematic breadth-first search is very helpful and restricts the search space a lot. With the help of PathFinder, developers easily understood how the failure came to be and were able to directly jump into erroneous state and behavior.

The last two hard failures required much more time with all tools and could not be solved in five debugging sessions including one with our Path Tools. We argue that these two failures are very hard to debug because the corresponding test cases fail far away from the failure-inducing defect. In the case of failure 5, all test-driven fault navigation participants were able to identify the defect within six to eight minutes, while standard debugging tools either could not solve this failure or need still more time. Failure 6 was so difficult that no developer could solve it with standard tools. With our Path Tools, two developers were able to find this failure after about twelve minutes. The remaining student came very close but could not solve it within 15 minutes. The same student already failed with standard tools at failure 5 but in this case he was also far apart from the root cause. Principally, PathFinder's assistance with following the infection chain backwards in combination with emphasized anomalies supported developers in isolating failure causes.

During debugging, we observed the participants and noticed some interesting insights. Developers with standard tools relied primarily on their intuition. Often, they guessed reasons for failure causes such as wrong behavior and infected state. In doing so, some developers had proper hypotheses about failure causes, but no developer was consistently better at guessing than another one. This observation was also reflected in the differences of required debugging time. In contrast, our test-driven fault navigation allowed developers to rely on a systematic debugging process and the advice of our tools. Developers linked the corresponding anomalies with their hypotheses and followed the infection chain backwards. All developers took advantage of the combined perspectives of our Path Tools. They usually started with a breadth-first search in PathMap and then followed the infection chain through suspicious behavior. As a consequence, the differences in debugging time were often marginal.

To assess the effectiveness of PathMap and PathFinder, we noted the point in time when developers switched from localizing suspicious system parts to debugging erroneous behavior back in time. For each failure, we marked this point in time with small lines in the corresponding columns of Figure 7.1. Usually, PathMap was applied for a breadth-first search in the first two minutes. During this time, developers obtained a first impression of anomalous system parts and they built first hypotheses about failure causes. In some cases (failure 1 and 4), these hypotheses were sufficient to localize defects without the help of PathFinder. In both failures, the defective method was ranked in a very suspicious anomaly (compare to Table 7.2) so that developers saw from the visualization alone where the defect might be located. However, these developers could not explain why defects result in observable failures. If defects could not be found directly, developers started PathFinder and followed the infection chain back. The required debugging time ranged from one to twelve minutes and strongly depended on the difficulty of failures and at which anomalous method developers opened a failing test case.

In summary, our test-driven fault navigation and its corresponding Path Tools are efficient for debugging because they allow developers to reduce the search space step by step until the root cause is found. With the help of our user study, we conclude that we are able to decrease debugging cost with respect to required time and developer's effort.

7.2.4. Feedback of Participants

After the user study, we interviewed the participants and asked them for critical as well as positive feedback on our approach and tools. All developers liked our Path Tools and conceived them as very valuable for debugging. A participant summarized it as follows: *"In particular for the debugging of non-trivial faults, I have the feeling that the failure localization requires less time. The classification of anomalies allowed me to create better hypotheses about the failure cause and to abbreviate the execution history."* Another student stated: *"The tools are fast and very well integrated with each other. The coloring of map and trace has helped a lot in focusing on suspicious entities."* A third developer mentioned: *"Compared to a standard debugger, I had no problem to take a look into a complete program execution and it was easy to understand how a failure comes to be."* Last but not least, one of them concluded *"I can very well imagine that the Path Tools improve debugging of our real failures, too."*

In addition to the positive feedback, they suggested usability improvements for our tools. They proposed promising new user interface elements and advanced query mechanisms for searching the call tree and its states. We will implement most of the recommendations in the future. Another point dealt with test-driven fault navigation as a new debugging method and its required change of thinking. They wished more practice with our Path Tools in order to debug even better. For this reason, we will make our approach available within our next software engineering course. Moreover, we will extend our upcoming debugging lecture with a test-driven fault navigation tutorial.

Finally, all developers confirmed that our approach had been promising for debugging with less effort. With our test-driven fault navigation, they received helpful advice for strengthening their hypotheses about failure causes and it appeared easier for them to follow infection chains backwards.

7.3. Effectiveness Study: Accuracy of Recommended Developers

To evaluate our team navigation, we assess the quality of our developer ranking metric by inserting numerous failures into a project, examining the accuracy of recommended experts, and comparing the results with a coverage metric that sums up authors of all methods executed by failing tests. If defects are unknown, our anomaly-based heuristic

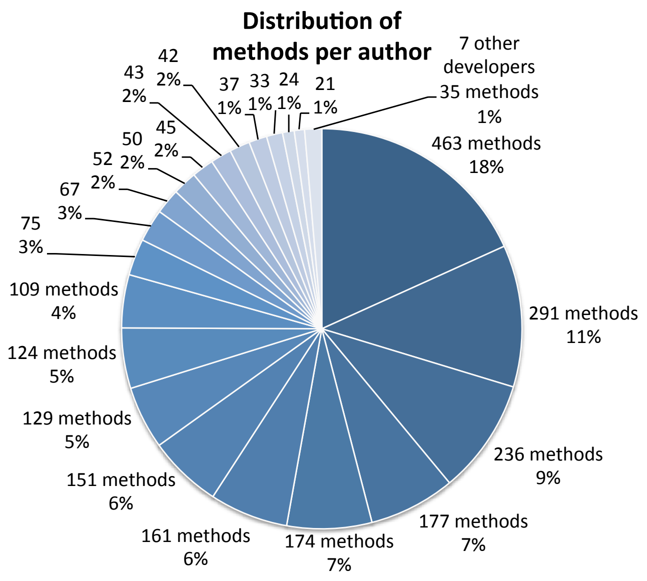


Figure 7.2.: Results for developer recommendation based on method distribution.

notably outperforms other approaches and proposes suitable developers with a probability of 80 % in the first five ranks.

To measure the accuracy of our developer ranking metric, we introduce a considerable number of defects into 4Conferences (see Section 7.1). We randomly choose 100 covered methods that are neither part of test code nor trivial (e. g., getters) and create a defect with a mutation engine[3] for each method. For example, we hide the method body and return the receiver object instead of executing the original functionality. We ensure the occurrence of each failure by observing corrupted test cases. For each of the 100 defects, we automatically run PathMap, compute a sorted list of recommended developers, and compare it with the *most active author* of the faulty method. In this evaluation, we consider such an author as the most qualified expert for understanding the specific defect.

Figure 7.2 shows the distribution of methods per author, where a method is always associated to its most active author. This distribution reveals that 4Conferences has been uniformly implemented by our 28 students. Only two developers have developed more than 10 % of the system, respectively 18 % and 11 %, and eleven developers have realized less than 2 % of all methods. If we randomly insert defects into 4Conferences, we have a chance of up to 18 % to pick the most qualified developer from the static distribution of expert knowledge. Furthermore, there is a maximum probability of 38 % to have the expert within the first three choices.

Figure 7.3 illustrates developer recommendations of a simple coverage metric that takes

[3]`http://www.hpi.uni-potsdam.de/hirschfeld/squeaksource/MutationEngine.html`

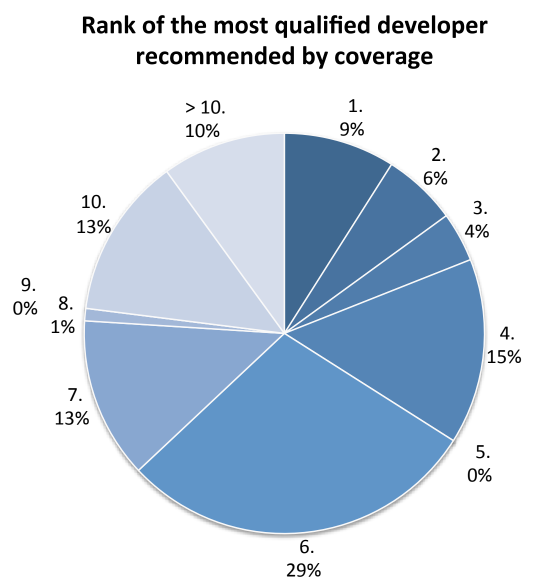


Figure 7.3.: Results for developer recommendation based on covered methods of failing test cases.

into account only covered methods of failing tests. This metric restricts the search space to erroneous related methods and sums up their authors. The chart shows the position of the most qualified developer in the list of recommended experts. In less than 20 % of all defects, the expert is found within the first three ranks. Compared to the static distribution of methods per developer, this value is below the 38 % of choosing an arbitrary expert. Only if we consider at least six recommendations and ignore the five false suggested experts, this metric seems to be better.

Figure 7.4 presents the position of the most qualified expert in the list of recommendations for our team navigation with its anomaly-based developer ranking metric. For almost a third of all defects, the responsible developer of the faulty method is ranked in the first place. With a probability of 60 %, we suggest the expert within the first three ranks and with a probability of 80 % in the first five ranks. As we restrict the search space to anomalies and their involved developers, our metric is able to outperform the two other approaches. For example, we achieve a probability of 60 % followed by 38 % of developer distribution and 19 % of the coverage metric for recommending experts in the first three ranks. Even if developers responsible for the failure 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 experts.

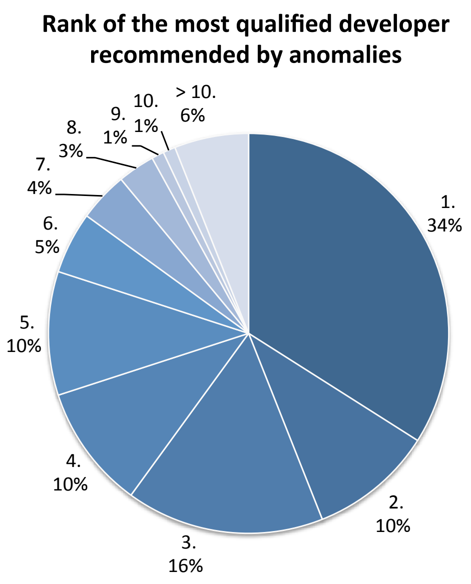


Figure 7.4.: Results for developer recommendation based on anomalies.

7.4. Effectiveness Study: Correctness of State Anomalies

As our inductive analysis harvests and generalizes common object properties from passing test cases, it is important to examine the quantity and correctness of these results for our state navigation. On the one hand, our harvesters need to determine as many generalized object properties from concrete objects as possible. On the other hand, falsely suggested object properties should be reduced to a minimum so that we only reveal true-positive state anomalies. For that reason, we compare our harvested object states with real used objects and discover that our inductive analysis results in promising recall and precision scores of almost 95 %. Thus, we reveal numerous helpful object properties whose mostly correct assertions support the emphasis of infection chains.

7.4.1. Experimental Setup

We measure the quality of harvested information by comparing two independent test suites of the AweSOM project (see Section 7.1). AweSOM possesses both unit tests, which verify the virtual machine functionality from the implementation perspective, and acceptance tests, which represent “real-world” scenarios. The acceptance test suite is written in SOM Smalltalk[4] and has been evolved over several SOM virtual machine implementations [102]. To ensure a high test quality, it checks the entire specification by

[4]SOM Smalltalk is a specific dialect that is used for all virtual machines of the SOM product family [102]. It is not directly executable by Squeak/Smalltalk.

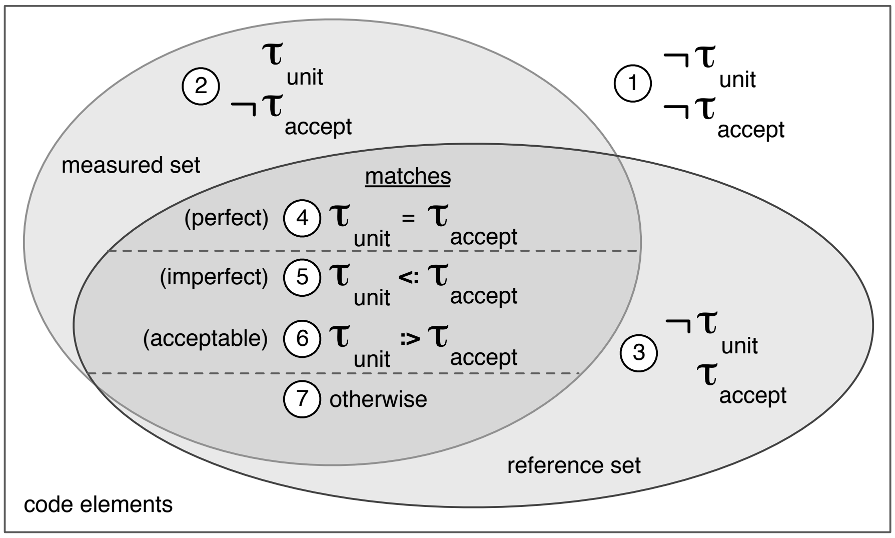


Figure 7.5.: Comparison sets for harvested object properties.

providing typical usage examples for specific features and actual runs of the complete SOM virtual machine. For our type and value range harvester, we execute both test suites at the same time and collect unit test object states (τ_{unit}) as the measured set and acceptance test object states (τ_{accept}) as the reference set. During harvesting, we consider only the system under observation and ignore test code for our inductive analysis. At each executed code element (meaning member variables, method arguments, and method returns) we collect the occurring objects and derive their common properties such as type information, size of collections, and spelling of strings. Comparing each object property in the measured and reference set leads to seven disjoint relationships that allow for a qualitative assessment of the specific harvester results. Figure 7.5 illustrates these seven relationships.

1. The code element was not covered by any test. No object information is available.
2. Only the unit test object (τ_{unit}) is available. No acceptance test has covered this element.
3. Only the acceptance test object (τ_{accept}) is available. No unit test has covered this element.
4. Both object properties are available and identical (perfect match).
5. Both object properties are available but τ_{unit} is more specific than τ_{accept} (imperfect match). This property has a broader scope in reality than covered by our unit tests. In other words, unit tests do not check the entire specification. In our state navigation, it might lead to contract violations such as types that are more general and collections that include more elements than expected.

6. Both object properties are available but τ_{accept} is more specific than τ_{unit} (acceptable match). In this case, the results are still acceptable because unit tests just verify and generalize more than required. For our state navigation, it indicates that more specific objects could occur at observed code entities; for example, `SequencableCollection` is harvested, and `OrderedCollection` or `SortedCollection` are used in reality.

7. Both object properties are not compatible to each other. For example, two different types that do not inherit from each other.

For good harvesting results, as much state information as possible must be collected (*coverage*) including most of the reference set (*recall*), and harvested object properties should be identical or at least related to each other (*precision*). We define the three metrics (coverage, recall, and precision referring to information retrieval [205]) based on the object property relationships of code elements as follows (numbers denote the sets S_i from above and illustrated in Figure 7.5):

$$\text{coverage} = \frac{|\tau_{unit} \text{ available}|}{|\text{code elements}|} = \frac{\left|\bigcup_{i=2,4,5,6,7} S_i\right|}{\left|\bigcup_{i=1}^{7} S_i\right|}$$

$$\text{recall} = \frac{|\tau_{unit} \text{ and } \tau_{accept} \text{ available}|}{|\tau_{accept} \text{ available}|} = \frac{\left|\bigcup_{i=4}^{7} S_i\right|}{\left|\bigcup_{i=3}^{7} S_i\right|}$$

$$\text{precision} = \frac{|\tau_{unit} = \tau_{accept} \text{ and } \tau_{unit} :> \tau_{accept}|}{|\tau_{unit} \text{ and } \tau_{accept} \text{ available}|} = \frac{\left|\bigcup_{i=4,6} S_i\right|}{\left|\bigcup_{i=4}^{7} S_i\right|}$$

Coverage is the extent to which the system includes harvested object properties. To maximize coverage, sets S_1 and S_3 must be minimized as they do not include state information from unit tests. *Recall* is the proportion of harvested (measured set) and used-in-reality object properties (reference set). To ensure a high value, set S_3 should be as small as possible. Last but not least, *precision* is the proportion of retrieved object properties that are perfect or acceptable matches. This metric evaluates the quality of state matches. For that reason, sets S_5 and S_7 should be small in comparison to sets S_4 and S_6. In particular, set S_7 should be nearly empty as properties contained therein are incomparable to each other.

Harvester		Type			Range		Both
Element	Member	Argument	Return	Member	Argument	Return	All
S_1	2	109	45	2	109	45	312
S_2	1	21	39	11	62	96	230
S_3	0	30	16	0	30	16	92
S_4	92	400	230	228	674	672	2,296
S_5	0	6	7	23	51	72	159
S_6	4	21	20	22	43	46	156
S_7	0	0	0	0	0	0	0
Coverage	97.9 %	76.3 %	82.9 %	99.3 %	85.7 %	93.6 %	87.6 %
Recall	100 %	93.4 %	94.1 %	100 %	96.2 %	98.0 %	96.6 %
Precision	100 %	98.6 %	97.3 %	91.6 %	93.3 %	90.9 %	93.9 %

Table 7.3.: Type and value range harvesting correctness. S_1–S_7 represent object state subsets (compare to Figure 7.5).

7.4.2. Harvesting Results

The results of our experiments are summarized in Table 7.3. It presents the absolute sizes of all harvested subsets and the computed values for coverage, recall, and precision[5]. The sets S_1, S_2, and S_3 show that some portions of the system are not or only partially covered. This was to be expected given a method coverage of about 82 %. Moreover, it can be seen that the parts covered by unit or acceptance tests *only* (sets S_2 and S_3) are small compared to the total. In particular, set S_3 has less than 100 entries meaning that only a few real used objects are not checked by unit tests. Set S_4, representing perfect matches, is the most important one, and also the largest. Sets S_5 and S_6, which have about the same low size, represent imperfect and acceptable matches. While acceptance tests (set S_5) require in some cases more generic types and larger value ranges, unit tests (set S_6) tend to be more fine-grained in that they check more alternatives than might be needed in "real-world" acceptance test scenarios. Looking at the detailed results, it can be noted that the same combinations repeatedly occur: for example, `OrderedCollection` and `SequencableCollection` (a test checks more than just the actually used collection), `False` and `Boolean` (a test does not cover one conditional branch), `nil` values (unit tests check robustness of methods), or collection size (acceptance tests base on larger "real-world" data sets). These cases are good suggestions for providing more and better tests. The set S_7 is empty. There are no conflicts between unit and acceptance tests to the effect that the system is not executed in a completely different way.

The coverage results in Table 7.3 allow our harvesters to extensively observe object states. AweSOM possesses with about 87.6 % a good unit test coverage that is a core requirement for our approach. The very high recall with 96.6 % shows that the unit tests largely

[5]Compared to type harvester's original evaluation [103], we present improved results because of additional unit tests that have been proposed by our test quality feedback approach later on [165].

cover the acceptance tests and thus the unit test quality is rather high. The remaining uncovered parts of AweSOM largely consist of code not used in reality; for example, legacy implementation artifacts or helper methods for debugging purposes. However, compared to the 82 % method coverage (see Table 7.1), our measured results are almost larger due to differences in computing coverage. For example, value range harvesting collects at one covered method more than one object state so that it may lead to a positive distortion in coverage[6]. Nevertheless, our coverage results support our harvesters in deriving valuable information for almost all important objects.

Recall and precision are crucial for evaluating harvesting quality, as they represent object properties used in reality, and the correctness of identified matches. The results obtained for these metrics are very good, at over 90 %. For *recall*, this means that most of the object states used in reality are actually found by unit tests. For *precision*, we can conclude that when an object property is harvested in a unit test, it will very likely occur in real code. This metric especially should be close to 100 %, as such a high value indicates to developers that harvested information is in accordance with reality. In other words, uncovered anomalies, contradicting harvested object properties, have a high probability of identifying corrupted state in the infection chain. We propose a false positive anomaly only with a probability of 6.1 %. As this value strongly depends on the quality of unit tests, we argue that it can be improved by providing more tests. For example, developers can compare harvested object properties with error-free usage scenarios and look for violated contracts. Although both metrics do not completely reach 100 %, they are very satisfactory with respect to our state navigation.

All three metrics—coverage, recall, and precision—should yield high values to ensure the quantity and correctness of our results. In particular, recall and precision should be close to 100 % as they best reflect the accordance between harvested information and reality. On average, for all harvested object properties, we obtain a coverage of 87.6 % (all possible object states), a recall of 96.6 % (overlapping of harvested objects with reality), and a precision of 93.9 % (perfectly and acceptably harvested properties), which underlines applicability and feasibility of our harvesting approach.

7.4.3. Emphasis of Infection Chains

With the help of our harvested object properties and their derived contracts, we finally evaluate our state navigation feature and its capability to emphasize infection chains. We add 50 failures to AweSOM and examine violated contracts between the observable failure and its root cause. We insert fifty times a defect with our mutation engine, run all tests, and analyze the violated contracts of all failing test cases. For each failing test case, we start our lightweight back-in-time debugger with enabled contracts and inspect upcoming state anomalies along the infection chain. During our experiment, the 50 non-trivial

[6] Without harvesting information, we do not know the exact number of potential object properties so that we have to mark such missing values in the corresponding sets (1) and (3) as only one.

	Invariant	Postcondition	Precondition
Type violation	83032	31646	7353
Size violation	6066	7	0
Range violation	720	297	203
Is or includes violation	1	44	4
Nil violation	0	272	4

Table 7.4.: Violation and contract types of all state anomalies in AweSOM.

failures lead to numerous state anomalies in long-running infection chains. Overall, we identify 763 failing tests cases with more than 129,500 contract violations. These failing tests include average traces of about 7,000 method calls and infection chains of about 1,750 method calls. On average, each mutated failure triggers 15.26 failing test cases and each test case includes around 169.92 state anomalies that emphasize 9.73 % of its entire infection chain. In the following, we look at the different types of state anomalies and their positions in infection chains.

Table 7.4 summarizes all state anomalies in AweSOM with respect to their contract and violation type. The most violated contract type is invariant with 69 %, followed by 25 % postcondition and 6 % precondition. Regarding the specific violations, there are 94 % of wrong type information, 5 % of oversized collections, 1 % of exceeded value ranges, and a few violated object properties. In other words, most state anomalies are violated type assertions in invariants checking instance variables. There are two reasons for the numerous unexpected types in AweSOM. First, Smalltalk is dynamically typed. There are no explicit type checks that prevent the use of wrong typed objects. Such failure causes are able to survive several method calls before an observable failure occurs. Second, AweSOM creates a lot of objects with a long lifetime that are required for building the complex virtual machine. If a defect causes a wrong typed object, it is often propagated across the entire building process. Since the object value is not accessed frequently, it does not trigger an observable failure. However, every time a specific building method is executed, it violates the corresponding invariant. On the one hand, this failure cause propagation and the many violated contracts emphasize large parts of infection chains. On the other hand, such type violations are often not expressive enough to completely understand the problem. But there are also other kinds of violations such as size, range, and object property. Even though these violations are not so numerous as types; they are more concrete and appear close by defects or observable failures. For these reasons, they help especially in understanding how the failure comes to be. In summary, it can be stated that the many type violations in invariants are adequate to follow infection chains backwards and the more specific value range violations are well-suited to explain failure causes.

Figure 7.6 shows the distribution and position of state anomalies of all 763 infection chains. Independent of the infection chain length, each percentage in the figure represents the relative position of a contract violation regarding its distance to the defect. For

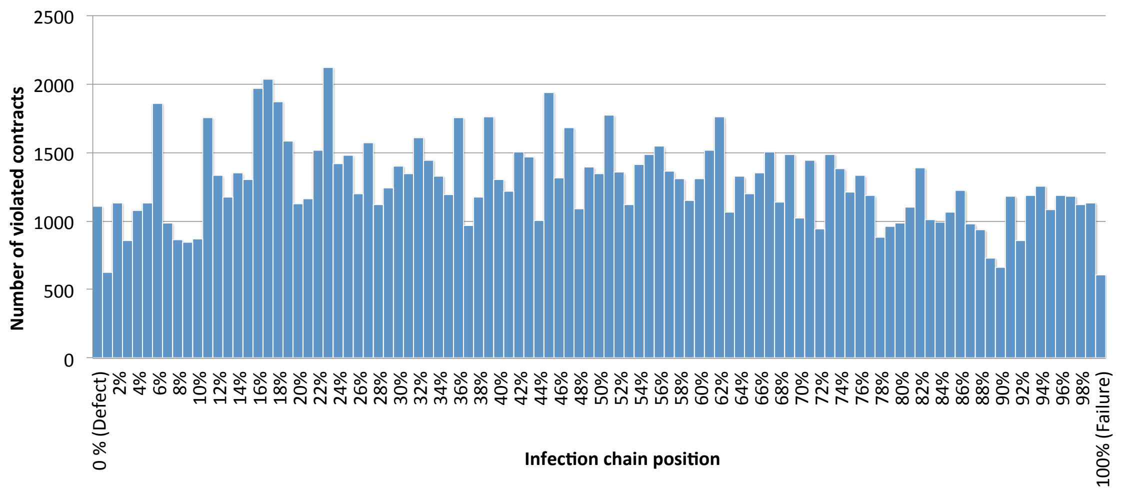


Figure 7.6.: Distribution and position of state anomalies in AweSOM's infection chains.

example, "2 %" relates to anomalies that we found close to root causes in the last 2 % of all method calls in their corresponding infection chains.

In total, we identify 129,500 contract violations that are evenly distributed over all infection chains. Each percentage of the relative infection chain in Figure 7.6 represents on average 17.5 method calls with 1,295 violations. This basically means that within 763 failing test cases we have 1.7 violations per 17.5 method calls. In other words, each 10th method call includes a revealed state anomaly and so emphasizes the infection chain. Thus, with the help of our state navigation developers can disregard 90 % of method calls and follow the evenly highlighted infection chain backwards.

Our dynamic contracts recognize fewer violations nearby defects and failures. If a failure occurs, the program execution stops and there is no possibility to check for violated contracts anymore. Regarding defects, they are usually inconspicuous and hard to detect with our contracts. Only 8 % of all infection chains have corresponding state anomalies directly at the defective method call. Although such anomalies are valuable, they are rare and their identification as root causes is not trivial because developers need to comprehend the relationships between all anomalies. For that reason, we argue that developers need to check entire infection chains to completely comprehend failure causes and their effects. In combination with our other test-driven fault navigation technique developers can browse complete execution histories and understand both failing program behavior and reasons for root causes.

Although we consider only 50 failures of one specific project preventing a general conclusion, we argue that our state navigation supports localization of failure causes because it evenly emphasizes state anomalies along infection chains and so allows developers to uniquely follow the flow of corrupted objects.

	Seaside	iCal-endar	4Confer-ences	Awe-SOM	Com-piler	zEmu
Creation of tree map (s)	23.56	1.17	1.79	1.03	1.13	1.31
Exec. time all tests (s)	6.20	1.05	198.57	5.06	0.91	6.26
Δ Fault localization (s)	5.46	1.33	8.64	9.00	1.85	7.78
Refined fault localization (s)	2.38	2.51	16.70	3.73	0.90	9.89

Table 7.5.: Average performance characteristics of PathMap in seconds.

7.5. Efficiency Study: Performance of the Path Tools Framework

We evaluate the performance overhead of our Path Tools framework by measuring the computation costs for collecting and presenting the required information for our test-driven fault navigation. From six typical and diverse Smalltalk projects (see Section 7.1), we analyze the average performance characteristics for our developer ranking metric and our three incremental dynamic analysis techniques. As a result, we keep response times and memory consumption low so that we are able to create an experience of immediacy when debugging with our tools.

We performed all experiments on a MacBook with a 2.4 GHz Intel Core 2 Duo and 8 GB RAM running Mac OS X 10.6.8, using Squeak version 4.2 on a 4.2.1b1 virtual machine.

7.5.1. Structure: Refined Coverage Analysis with PathMap

We measure the run-time overhead for the structure navigation with our PathMap and its refined coverage analysis by analyzing all tests of our six studied projects. We do not insert failures because the analysis overhead is independent of their occurrence. PathMap executes the entire test suite with and without fault localization. We refine fault localization at statements only at non-trivial methods including a McCabe complexity [147] greater than one. So, we exclude simple methods with sequential behavior where all statements have the same suspiciousness score. The average results for the performance of PathMap are described in Table 7.5.

The first row presents the one-time cost (in seconds) of analyzing the source code of the software system and creating the tree map view. Although this time is just about 1 second for mid-sized projects, in the case of large projects such as Seaside it can take a significant amount of time (23 seconds). However, this creation is done only once, when the PathMap tool opens. We already know that our current implementation tends to be slow because it creates for each source code entity a separate user interface morph object. In the near future, we will improve this performance by drawing the tree map within a single object.

	Seaside	iCalendar	4Conferences	AweSOM	Compiler	zEmu
Computation Cost (s)	11.19	5.00	7.92	3.44	4.49	3.76

Table 7.6.: Worst case computation cost for our developer ranking metric in seconds.

The second row shows the pure run-time for executing all tests. While five projects run their test suites in less than ten seconds, the 4Conferences project requires more than three minutes due to several slow running acceptance tests. These tests check complete user workflows including multiple interactions with the Web application with the help of the Selenium test framework[7] .

The third row presents the overhead resulting from spectrum-based fault localization. PathMap's fault localization slows down execution by a factor of 1.9 for Seaside to 2.9 for AweSOM. In the case of 4Conferences, long waiting times for responding to interactions of acceptance tests explain the minimal slow down factor of 1.04. In all other cases, the variation originates from additional instrumentation and visualization costs. Nevertheless, this overhead is low enough for applying spectrum-based fault localization frequently.

Finally, the fourth row reveals the time for refining fault localization at statements. The time mostly depends on the number of covering tests per method and their corresponding run-time. In most cases, developers receive refined results for complex methods in less than three seconds (without 4Conferences's long running acceptance tests the 80th percentile is below 2.6 seconds). We argue this time is still acceptable compared to a complete statement coverage analysis that possesses a slow down factor of about 100 in Squeak/Smalltalk [103].

7.5.2. Team: Developer Ranking from Source Code Repositories

To measure the computation cost of our developer ranking metric, we assume the worst case by declaring each method as anomalous and computing the most active developer. We expect that each method of a project has a suspiciousness and confidence value of one. Thus, our metric has to gather all method commits in order to determine expert knowledge. To compute an average run-time, we repeat the ranking ten times for each project. Table 7.6 presents the worst case computation cost for our developer ranking metric.

For computing our metric, we require between 3.5 seconds for AweSOM as the smallest and about 11.2 seconds for Seaside as the largest system. The computation cost strongly depends on the number of analyzed methods. For each method, our metric requests Smalltalk's version control system in order to obtain all changes and to compute the most

[7]`http://www.seleniumhq.org/`

	Seaside	iCal-endar	4Confer-ences	Awe-SOM	Com-piler	zEmu
Calls/Tests	327	434	2,326	6,664	4,502	1,178
Exec. time per test (ms)	0.76	3.34	1910.82	17.33	7.69	15.12
Δ Shallow (ms)	336.17	258.16	281.02	235.79	247.23	172.28
Shallow memory (kbyte)	93	102	558	1,464	991	259
Δ Refinement (ms)	16.92	2.67	19.65	5.93	2.15	1.81

Table 7.7.: Average performance characteristics in milliseconds and kilobytes for PathFinder, our lightweight back-in-time debugger based on step-wise run-time analysis.

active author. In particular, accessing method changes slows down the execution as it includes excessive I/O handling. Considering our worst case scenario that each method is an anomaly, the mentioned computation cost reflects the maximum per project. In a realistic setting, anomalies cover only a subset of all methods and so we argue that the computation cost is still acceptable. For example, in our Seaside typing error example we return the result of 54 anomalous methods in about 3 seconds.

7.5.3. Behavior: Step-wise Run-time Analysis with PathFinder

We measure the run-time characteristics of our lightweight back-in-time debugger by running each test of a project with PathFinder and analyzing the overhead produced by our step-wise run-time analysis. For each single test, we record the total time and memory needed for collecting the required data and rendering test behavior. While our shallow analysis determines the entire method call tree, our refinement analysis creates a deep copy of the returned object of a random call node. Table 7.7 describes the average performance characteristics of PathFinder.

The first row lists the average number of method calls per test for application code that is instrumented for shallow analysis. This value relates the following performance measurements with the size of call trees. In our larger step-wise run-time analysis evaluation, we show and discuss that the time and memory overhead grows linearly with respect to the number of methods that are invoked by a test [168].

The second row lists the average and pure run-time per test case. As 4Conferences' acceptance tests still include high run-time cost, immediate feedback from Path Tools is hindered. Apart from that, in all other projects executing a single test case requires less than 18 milliseconds on average.

The third and fourth row demonstrate the overhead associated with building the lightweight method call tree. On average, the run-time overhead is between 170 and 340 milliseconds meaning that the shallow analysis is quite fast. The 99th percentile for this value is below

750 milliseconds. In the case of Seaside, which has the lowest number of called methods but the highest performance decrease, we discover that the one-time instrumentation of all 5,500 methods is rather expensive and requires around 250 milliseconds alone. The collected data during shallow analysis required less than 320 kilobytes on average and is directly related to the size of call trees.

The last row deals with the refinement analysis that allows developers to reload state information on demand. In doing so, the required analysis overhead only depends on the effort for creating a deep copy. Thus, our refinement analysis also imposes minimal cost because the 95th percentile is below 25 milliseconds for all test cases.

Our empirical results illustrate the feasibility of our interactive approach: The imposed overhead for dynamic analysis is distributed across multiple test runs and allows for short response times. For all projects, the time it takes to collect data for generating a call tree is below 400 milliseconds and for a refinement step below 140 milliseconds on average (this includes the time it takes to run the test). While our evaluation focusses on measuring the time required for our incremental dynamic analysis, we also conduct independent experiments to consider the response time of PathFinder's graphical user interface. The time required for rendering call trees averaged around 200 milliseconds for the AweSOM project, which involves the highest number of calls per test. We argue that this supports our claim of achieving immediacy characteristics by providing a fast visualization of run-time information. Schneiderman [190] argues that two seconds is the upper limit for responding to a user request. PathFinder and our step-wise run-time analysis support these fast response times when debugging back in time because run-time data can be provided in considerably less than two seconds in the majority of cases [168].

7.5.4. State: Inductive Analysis with Harvesters

We evaluate the efficiency of our inductive analysis and state navigation by measuring the performance of all six projects during harvesting and verifying contracts. We start with quantifying the required time for harvesting type and value range properties. Then, we look at the creation time of contracts. Finally, we consider the execution overhead for checking assertions while running test suites with activated contracts. Table 7.8 summarizes these performance characteristics.

The first row presents the pure execution time of complete test suites again. This allows for a better comparison of the following measurements.

The second and third row show the overhead associated with harvesting. Both harvesters have different execution overheads per project, which range from 5 seconds up to 90 minutes. This is largely because harvesting strongly depends on the existing object space. In particular, nested objects and large collections require a vast amount of time for analyzing their properties. For example, in collections each single element has to be explored for finally generalizing children's data. While iCalendar has mostly shallow

	Seaside	iCalendar	4Conferences	AweSOM	Compiler	zEmu
Exec. time all tests (s)	6.2	1.1	198.6	5.1	0.9	6.3
Δ Type harvester (s)	494.1	4.8	199.4	4,668.9	359.1	33.3
Δ Value range harvester (s)	39.6	82.9	252.1	844.4	44.0	112.7
Contract creation (s)	43.5	9.9	11.6	13.2	15.3	15.9
Δ Contract validation (s)	66.8	17.7	28.6	83.4	33.2	41.2

Table 7.8.: Average computation costs in seconds for our inductive analysis, contract creation, and contract validation.

objects without deep structures, AweSOM has many dictionaries that comprise numerous parameters of the SOM source code and the virtual machine. To solve this problem, we provide optional run-time optimizations in return for less precise information. For example, we are able to limit the inductive analysis to selected elements of collections. With the assumption that collections include only one object type, this loss of precision is acceptable at least for our type harvester. In the case of AweSOM's types, we can decrease the required harvesting time from 4,668 seconds down to 98.9 seconds.

The fourth row deals with the one-time cost for creating source code contracts from already harvested information. This time ranges from 10 to 45 seconds per project and mostly relates to the number of methods that have to be recompiled with assertions.

The last row considers the overhead for checking numerous assertions while running all tests with activated contracts. On average, derived assertions slow down the execution by a factor of 15. However, there are also a maximum factor of 37.9 for Compiler and a minimum factor of 1.1 for 4Conferences. In the first case, the Compiler project includes both a large number of assertions and method calls. In the latter case, 4Conferences owns several acceptance tests with several waiting times that restrict the influence of assertions.

Due to the partial slowness of our state navigation, we provide several optimizations for speedup harvesting and reducing cost of checking contracts. We allow developers to select the levels of detail to improve harvesting. They are able to decide about object properties such as specific types and value ranges, analyzed code elements, and the precision of generalized results. With the help of these options, developers can customize our harvesters according to their needs. Nevertheless, we consider the investigation of quicker implementations an important area of future work. For example, it is not necessary to always run a full test suite to obtain changed run-time information. Integrating our inductive analysis with continuous selective testing tools [186, 196], which execute tests only if they cover source code changes, is a worthwhile direction. Regarding the execution cost for checking assertions, we suggest to apply our state navigation only for a single test case and not entire test suites. This would slow down the shallow analysis of our step-wise run-time analysis by a factor of 15. For that reason, we split the state navigation into an additional test run that checks contracts in background. Thus, developers can still immediately inspect run-time behavior of a failing test case and as soon as we finish the validation of contracts we highlight the failure's infection chain.

7.6. Threats to Validity

Our evaluation setting has several characteristics that might limit validity.

In general, the Smalltalk context of our evaluation might impede validity by limited scalability and general applicability. However, the Seaside Web framework is a real-world system and it exhibits source code characteristics comparable to particular complex Java systems such as JHotDraw [168]. Even if the remaining projects are only mid-sized systems, 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.

We rely on tests to obey certain rules of good style; they should be reproducible and 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.

Our *user study* only considers one particular project, its six failures, and undergraduate students. We consider the iCalendar project as a real-world application because it is mature and an important part of several other systems. The six failures are realistic and related to similar known defects that we have found in other projects. We treat our students as professional developers because of their longstanding programming experience. Although we require a larger study for a general conclusion, our user study already reveals the benefits of our approach and its tool suite.

The chronological order of debugging the first three failures with standard tools could positively influence participants' program comprehension. There is a chance to localize the remaining three failures with our Path Tools more simply. To reduce this factor, we have a preparation phase of two hours to become acquainted with iCalendar. During this time, developers read not only source code but also applied PathFinder to understand behavioral examples of test cases [168]. Furthermore, we make sure that all defects and their infection chains are unique. They are located in completely different system parts and their failure-reproducing test cases do not overlap each other.

Regarding our *effectiveness study*, we limit the evaluation of our developer ranking metric and the emphasis of infection chains to mutated defects only. We have not yet checked the effectiveness in a realistic setting due to difficulties in finding suitable Smalltalk projects. Many projects suffer from an unbalanced distribution of method authors, missing tests that reproduce failures, or failure reports that are not related to source code changes and vice versa. For these reasons, we plan a more controlled experiment with the 4Conference system as part of our next software engineering course. If students find a failure, our student assistants, which are former developers of 4Conferences, are supposed to write a reproducing test so that we can verify our metrics. Even though we evaluate our heuristics only with quite a number of synthetic failures, we argue that their results are already satisfactory with respect to recommended developers and highlighted infection chains.

Our failure inducing mutations cannot cover all kinds of defects. As we base our mutation engine on previous work [46, 127], our studies still involve a large number of possible defect types. These synthetic defects are realistic problems such as conditional faults, wrong computations, or missing side effects. Due to the high number of considered defects in our studies, we can ensure a broad distribution of most different failure types all over mutated applications.

The recall and precision metrics of the correctness of harvesting results could easily be manipulated to reach 100 %. For example, in type harvesting all code elements could be assigned the type `Object` in order to reach a complete recall and precision. However, such "optimizations" would result in acceptable matches for almost all generic elements. Our set S_5 includes only 6 % of all matches meaning that it does not influence the metrics negatively. For the sake of completeness, we would like to point out that `Object` was harvested as least specific type in only 2.3 % of all cases. Upon closer observation, these cases are unproblematic, for example, when elements in collections may have arbitrary types.

Regarding the quality of violated contracts in the emphasis of infection chains, we expect no false positive state anomalies. We consider each anomaly as valid with respect to the root cause because of our experimental setup. Each failing test case previously contributed as passing test case to harvested object properties. Without a mutated defect and activated dynamic contracts, all test cases pass and no state anomaly occurs. Thus, each contract violation in consequence of a mutation is caused by this defect.

Finally, in our *efficiency* studies we disable garbage collection during measurement to be able to gather memory consumption data, and to elide performance influences of garbage collector runs. In a realistic setting with enabled garbage collection, minimal slowdowns would be possible.

7.7. Summary

We evaluated test-driven fault navigation with respect to its practicality for debugging, the effectiveness of our automatic test-driven heuristics, and the efficiency of our Path Tools framework. First, we conducted a user study to evaluate our entire approach with respect to the reduction of debugging cost. As a result, we found out that developers who applied our Path Tools required less time and effort for debugging in the majority of cases. Second, we analyzed the accuracy of the developer ranking metric for our team navigation by assessing recommended contact persons for numerous failures. With a probability of 60 %, we suggested the expert developer within the first three ranks and with a probability of 80 % in the first five ranks. Third, we considered the correctness of harvested state anomalies with the comparison between test-based and real-used objects. Apart from the fact that derived object properties matched real used objects with a recall and precision score of almost 95 %, our state navigation especially utilized this information to evenly emphasize corrupted state along infection chains. Finally, we measured the performance characteristics of our Path Tools framework. The implementation of our test-driven fault navigation and incremental dynamic analysis came with a low overhead factor that enabled an immediate experience when debugging with our tools.

Part IV.

Related Work and Conclusion

8 Related Work

In this chapter, we present testing and debugging approaches that are related to our test-driven fault navigation. In testing (Section 8.1), we consider new perspectives, namely tests as examples and their hidden knowledge. In debugging (Section 8.2), we discuss methods to localize failure causes in general and for each fault navigation step in particular. In addition to these topics, we further present specific software visualizations supporting debugging activities (Section 8.3) and related techniques for our incremental dynamic analysis (Section 8.4).

8.1. Testing

In general, software testing is the process of executing a program with the intent of finding failures [155]. Testing evaluates the reliability of an implementation and reveals discrepancies in expected program behavior. With the help of this valuable feedback, developers are able to identify and correct code-specific problems early on. Nowadays, testing is an integral part of most development projects and it is strongly recommended to deliver high quality software [3][1]. In agile methodologies, testing is a cornerstone for successful software development [28]. It allows developers to react on the steady changes in requirements and technologies by offering a safety net that catches unavoidable mistakes during development. For example, test-driven development [27] encourages writing of unit tests before programming and so ensures a high quality of both tests and code.

Unit testing [98] provides the foundation for our test-driven fault navigation and is one of the most common testing technologies. After writing a failing test case that reproduces the observable failure [219], we compare all the other unit tests with it, reveal anomalies, and follow its infection chain backwards. For that, our approach ideally requires a large number of unit tests that cover most parts of the system. In recent years, the quality and quantity of unit test suites has been investigated by a number of studies. In 2005, several well-tested Smalltalk projects focussed their unit tests on executing just a single method [83]. These tests check API calls and limit their scope to one specific method under test. However, the style of unit tests varies depending on the corresponding

[1]This reference by the “Federal Association for Information Technology, Telecommunications and New Media (Bitkom)” describes guidelines for the industrial software development in Germany.

developers because of missing guidelines. In 2007, unit testing was already widespread but its development lacked effectivity because of still used ad-hoc methods [32]. In 2009, a systematic survey and classification of unit testing techniques further confirmed its importance and also reported development improvements [208]. Finally in 2012, a qualitative study investigated testing in plug-in based architectures and concluded that little testing takes place beyond the unit level [89]. All these studies confirm that unit testing is a well-known and common technique. For that reason, we argue that many software projects comprise large unit test bases and so fulfill the requirement of our approach.

In the following sections, we present related work with respect to our test-driven fault navigation that leverages test cases as entry points into reproducible behavioral examples and valuable source of hidden knowledge.

8.1.1. Tests as Examples

As we consider test cases as small examples into behavior, we are different from the typical testing perspectives that are mostly interested in revealing failures. Apart from our perspective on test cases, there are also similar point of views by other approaches.

Example-centric programming [66] proposes to understand source code and its abstract concepts by thinking of concrete examples. It introduces a first prototype for the Eclipse development environment called example-enlightened editor. When needed, developers can benefit from this editor by seeing examples directly at the corresponding source code. The approach concludes with the suggestion of unit tests as source of examples but it does not further investigate this idea.

The *metaphor of examples* guides a thesis that deals with unit testing for assuring quality and documentation of methods [82]. The approach suggests making the implicit dependencies of unit tests more explicit and so to integrate them into the development environment. It provides the Eg meta-model in order to sort, classify, and decompose unit tests meaningfully. With the help of this meta-model, the *EgBrowser* [84] offers an interactive editor for developing and composing higher-level tests from examples produced by other tests. In doing so, the tool maintains the explicit links between tests and units under test and so allows developers to reuse these links for debugging, documentation, and coverage analysis. Such explicit links can also help to improve testing frameworks such as JUnit. The *JExample* [131] extension allows developers to declare dependencies between test methods in order to optimize the organization of running tests. For example, as the framework knows the dependencies between tests it can prevent the execution of unnecessary failing test cases.

Another approach recommends to *embed examples as part of a new software activity* [25]. Although it proposes a collection of techniques for writing examples, the reuse of unit tests is not part of them. One example of such a technique with unit tests could be a tool that *automatically derives test cases from read-eval-print loops* [158]. In functional programming, developers typically implement functions with such loops and so they can be seen as ideal examples for documenting library interfaces. Unfortunately, this approach is not easily applicable to imperative programming languages.

A preliminary study analyzes the benefits and drawbacks of unit tests as examples for program comprehension [202]. On the one hand, unit tests including bad smells such as checking multiple concerns are hard to understand [203]. On the other hand, unit tests with a focus on a single method may help developers in comprehending abstract source code.

Compared to our test-driven fault navigation and in particular our early *debugging into examples* technique [197], we consider test cases completely as behavioral examples. They offer reproducible entry points, their coverage links tested methods, and their re-execution allows developers to immediately access concrete run-time information. In particular, our PathBrowser connects this exemplary knowledge to source code and assists developers in program comprehension.

8.1.2. Applications of Hidden Test Knowledge

Test cases also yield a valuable source of hidden knowledge for multiple software maintenance activities. Apart from our test-driven fault navigation that applies this finding for debugging, there are other approaches that further improve software quality, documentation, and program comprehension.

The measurement of test coverage is an indicator for software quality [110]. The more source code entities such as methods, branches, and statements are executed by test cases, the higher is the test quality that ensures the realization of requirements and confidence in source code. As this metric can automatically be computed, there exists at least one test case coverage tool for almost each programming language [213]. They mostly differ in granularity, performance, and unique features such as test selection, automatic test case generation, and customization of test reports. Furthermore, the visualization of collected test coverage assists developers in perceiving important relations between tested software components. For example, *Hapao* [13] presents test coverage in the form of polymetric views [132]. It visualizes shapes of methods and classes, combines them with additional metrics, and indicates where more testing is required. Also, our *test quality feedback* is a helpful visualization that extends PathMap to improve effectivity and efficiency of unit testing [165].

Test cases also keep the system documentation up to date. For example, agile processes consider tests as living documentation that describes small usage examples of interfaces, libraries, and frameworks [204]. As long as they do not fail, they offer a valuable source of information about the inner workings of a system. Their analysis can also help to create additional documentation. For example, they automatically reveal observer abstractions [212] and generate sequence diagrams [52]. Regarding program comprehension and documentation, test cases can also be used for establishing traceability links between arbitrary software artifacts and source code entities [206]. The recovered traceability supports several software development activities such as impact analysis, test case selection, and software understanding. One approach to establish traceability links is to connect test cases with specific concerns and to follow their execution paths [111]. For example, we apply this method in a previous approach to recover source code entities that implement a specific use-case concern [105]. With the help of annotated acceptance tests, we follow which objects participate in the execution of use-cases.

The broadest approach that recognizes test cases as more than just error finders is Bellcore's χ*Suds* [6]. By combining control graphs, traces, and slices, five tools mine system tests to aid in understanding, debugging, and testing programs. χ*Vue* locates features and their interactions. χ*Prof* identifies performance bottlenecks. χ*Atac* determines how code can be tested better. χ*Slice* pinpoints errors by comparing multiple slices being selected by programmers. χ*Regress* chooses proper test cases for regression testing.

8.2. Debugging

As software includes failures, developers have at all times experienced lengthy and laborious correction tasks [140]. In the majority of all software failures, the human error, the increasing complexity of software [97], and the lack of debugging knowledge [138] are largely responsible. Studies confirm these observations by investigating characteristics of software failures, their reasons, and solutions. In open source projects, bug databases report that above all semantic defects are common and difficult to correct [136]. That is because of their application-specific nature, which requires a thorough understanding of the program. However, the less knowledge developers have, the higher the probability of introducing difficult defects. This conclusion is also reported in other studies that consider the rate of defects in internal APIs [162] and the number of post-release failures due to incorrect fixes [215]. From a more fine-granular view, a human study investigates the kinds of defects that are hard to locate [79]. As a result, developers meet problems especially with missing statements and complex data structures including large cognitive demands. Another major reason for challenging failures is the delocalization of required information in source code [65]. Developers have to understand infection chains in little-known system parts because failure causes are distributed throughout the entire program. Furthermore, object-oriented programming concepts such as late binding make the tracing of causes and effects even worse.

For all these reasons, general debugging methodologies should assist developers in the comprehension of failure causes and their effects [65, 138, 140]. *Debugging by thinking* [150] explores methods from the perspective of intellectual disciplines. It proposes to debug like a detective, mathematician, or engineer in order to ask the right questions for proper failure cause hypotheses. Independent from tool support, each intellectual perspective describes a systematic procedure to follow infection chains backwards. *The developer's guide to debugging* [90] shares experience reports for the most frequent real-world problems. In the case of similar failures, the proposed best practices give advice to solve difficult problems. The *traffic* principle [219] summarizes the state of the art in debugging. Starting with *tracking* the problem up to *correcting* the defect, it provides a systematic procedure from the first failure report to localization of the root cause to its fix in source code.

In the following, we discuss more specific debugging techniques that are related to our test-driven fault navigation and its individual steps.

8.2.1. Scientific Debugging

Our test-driven fault navigation is based on the concept of the scientific method, which originates from the natural sciences [211]. In general, this method allows scientists to develop and examine a theory that explains and predicts observations. In debugging, the scientific method allows developers to find a diagnosis for an observable failure [219]: developers observe a failure and create a hypothesis; they make predictions and test their hypothesis; they refine or reject the hypothesis and repeat these steps until a diagnosis is found. The most challenging part of this method is to create, evaluate, and refine proper hypotheses. For that reason, there are several debugging tools that partially support the scientific method in a variety of ways. They can be classified into *deduction*, *observation*, *induction*, and *experimentation* techniques [218]. Deduction analyzes source code from a static point of view, observation considers one concrete execution path, induction derives generalized information from concrete values, and experimentation finds failure causes in a controlled manner. Even our test-driven fault navigation conforms to this classification: experimentation with the scientific method; induction with anomalies; and observation with back-in-time debugging.

Different from a manual debugging session with the scientific method, *algorithmic debugging* is a semi-automatic approach [193] that interactively guides developers along infection chains. It systematically creates hypotheses asking developers about possible infections. Depending on their answers, it refines hypotheses and isolates failure causes step by step. There are implementations for the logic programming languages Prolog [193] and the imperative language Pascal that realize the concept with the aid of program slicing [78]. Furthermore, declarative debugging [157] generalizes algorithmic debugging for diagnosing failures in arbitrary programming languages. However, the general approach does not scale with the increasing complexity of current programs [219]. Developers have to answer either too many or too generic questions.

8.2.2. Spectrum-based Fault Localization

As our structure navigation adapts spectrum-based fault localization, we first consider related approaches. After that, we look at current improvements and empirical studies that verify and optimize the effectivity of revealed anomalies. Finally, we present fault localization techniques that not only consider coverage but also state spectra.

Spectrum-based fault localization is an active field of research where passing and failing program runs are compared with each other to isolate suspicious program entities. To support the identification of anomalies and finally failure causes, *program spectra* summarize the required run-time data such as executed program statements [101]. With the help of this information, several tools restrict the search space for localizing failure causes. *χSlice* [9] subtracts execution slices of passed test cases from an execution slice of one failing test case. The difference of all slices, also called dice, may contain the root cause. *Tarantula* [122] analyzes covered statements with respect to all passed and failed test case executions and visualizes the overlapping behavior according to their results. At the system overview level, each statement is represented as a line of pixels and colored with a suspiciousness score that refers to the probability of containing the defect. Later on, *Gammatella* [160] presents a more scalable and generalized visualization for monitoring deployed software systems. In form of a tree map, the tool consolidates all system classes and how they continuously interact with each other. It does not deal with test cases but rather with the exceptional behavior of running systems. The *Whither* tool [180] collects spectra of several program executions that are then classified by users as either correct or not. According to the nearest neighbor distance criterion, it determines the most similar correct and faulty runs, and creates a list of suspicious differences. *AskIgor* [49, 217] also identifies state differences of passed and failed test runs and automatically isolates a subset of program states that lead to the failure. To follow the infection chain back to the failure-inducing defect, it combines delta debugging [220] with cause transitions. While the first isolates relevant states by systematically altering execution and comparing differences of passing and failing runs, the latter focuses on when and where causes originate. A first empirical study [121] comparing these different spectrum-based approaches concludes that the Tarantula technique is more efficient than the other ones.

Several other approaches have improved the concept of spectrum-based fault localization. *Lightweight defect localization* [55] looks at method call sequences per object/class instead of single statements. *Failure-inducing chops* [95] first minimize failure-inducing input with delta debugging and then intersect the corresponding forward dynamic slice with the backward dynamic slice of the erroneous output. *Debugging in Parallel* [120] clusters failing test behavior targeting the same fault and allows for debugging these specialized test suites simultaneously by multiple developers. The *time-spectrum-based fault localization* [214] searches for exceptional run-time deviations regarding the behavioral model of passed test cases. *Zoltar* [113] introduces a low-cost Bayesian metric to localize multiple faults and fault screeners that early interrupt program execution with violated invariants. The *Apollo* tool [16] automatically generates test cases for Web applications and combines different Tarantula algorithms to identify execution and HTML failures very successful. Finally, *Falcon* [163] localizes faults among threads by monitoring memory-access sequences, detecting data-access patterns, and reporting their relation to passing and failing program runs.

There are also a few empirical studies verifying and optimizing the effectivity of spectrum-based fault localization. The first experiment [216] investigates the influence of test suites, their compositions, and possible reductions. It concludes that efficiency strongly depends on how to reduce and choose test suites. Another study [117] with a similar focus on test case prioritization concludes that random ordering of test cases can also be efficient in localizing faults with less effort. The next experiment [189] shows that different kinds of faults require different types of coverage. It presents a new combination of statement, branch, and data dependency coverage that leverages the unique properties of each coverage type. A more comprehensive study [2] about the impact of metrics on the diagnostic accuracy states that similarity metrics are largely independent of test design and that the Ochiai coefficient consistently outperforms all other approaches. Moreover, they show that the near-optimal accuracy, being at filtering 80 % of system parts, can be obtained with a few instead of all test cases.

Apart from coverage spectra for revealing anomalies, there are other approaches that further analyze parts of program states. *Statistical Debugging* [137], also known as scalable statistical or cooperative bug isolation, analyzes automatically created bug reports from deployed software in order to isolate root causes. The corresponding *Holmes* tool realizes this approach and provides an iterative bug-directed profiling that pinpoints the likely failure cause over multiple executions by instrumenting only statistically relevant suspicious program entities in the field. It monitors successful and failed runs via predicate profiles which represent specific program properties such as exceptional behavior, unused return values, and scalar pairs. With the limitations to just a few program points and properties, this approach ensures affordability in an industrial setting. Later on, the Holmes tool also collects path profiles to find execution paths that correlate with failures [47]. Further improvements to statistical debugging include: the *categorization of relevant predicates* without any prior knowledge of program semantics [143]; the *automatic mapping of predicates to faulty control flow paths* [118]; the combination of several statistical techniques to *estimate causes and effects* [18]; and the adaption of *monitoring depending on control-dependence graphs* [17] Based on the idea of predicates, another approach [222] identifies anomalies by *altering predicates* and so possibly correcting the control flow. In this case, the algorithm detects a critical predicate with a high probability of a failure cause. Analogously, *value replacement* [114] automatically alters entire states used at statements of a failing run in such a way that incorrect output becomes correct. Located statements are either failure causes or close by.

All presented approaches produce anomalies in the form of 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 with the help of already gathered suspiciousness information and supports developers in further approximating the root cause. Principally, our approach adapts the Tarantula technique at the method-level with the Ochiai similarity coefficient and so ensures satisfying results within short response times. Our PathMap is directly integrated into the unit test framework, provides a lightweight coverage analysis, visualizes the suspiciousness data in form of a scalable tree map, and recommends suitable developers as contact persons. Our PathFinder classifies run-time behavior by reusing the already collected information and, if desired, refines fault localization at statements. Thus, we combine spectrum-based fault localization data in different perspectives and so give advice to developers on how to follow infection chains back to their root causes.

8.2.3. Determining Developer Expertise

Our team navigation with its developer ranking metric is mostly related to approaches that automatically determine expert knowledge for development activities. In particular, we discuss the identification of experts for specific source code artifacts and the assignment of bug reports to developers.

The *expertise recommender* [148] proposes a general architecture allowing the administration of user profiles for arbitrary collaborative development activities. As an example, they instantiate their framework for the implementation activity of a real-world software company. In doing so, they figure out the fundamental "Line 10 rule"[2] that recommends expertise by considering the change history of source code. Developers who last modified a specific source code artifact have the most up to date knowledge. The *expertise browser* [154] automatically quantifies people with desired knowledge about source code by analyzing still more information from change management systems and the program's history. Developers collect experience atoms for specific system parts where they have recently fixed a problem or enhanced a feature. From these expertise profiles, other stakeholders identify individuals with a broad expertise in specific system parts. *Ownership maps* [86] present a compact overview for all files and their evolution with respect to corresponding expert knowledge. With the help of CVS logs, it analyzes the commit history, identifies the file ownership, and visualizes the results. Among others, these maps help to answer overall questions such as which author developed which part of the system at what time. *XFinder* [123] 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. On the supposition that people who have substantially contributed in the past are likely the best for future changes, the tool identifies experts for specific projects, packages, and files. The accuracy of the first three recommended

[2]The name corresponds to the position of author credentials in change log messages.

developers is between 43 % and 82 %. The *emergent expertise locator* [152] approximates, depending on currently opened files and their histories, a ranked list of suitable team members. All of these approaches rely on the assumption that programmer's activity indicates some knowledge of code. An empirical study [76] about the "Line 10 rule" investigates the frequency and recency of source code interactions. As a result, it confirms the rule and presents additional factors that also indicate expertise knowledge such as authorship or performed tasks. With these findings, subsequent approaches enhance the results of pure history information with *usage expertise* [191], which also considers the application of API methods, and *source code familiarity* [77], which additionally analyzes developers' interaction with code.

There are also specific *bug triage* techniques that focus on the analysis of bug and source code repositories to automatically assign bug reports to the most qualified developers. A first bug triage approach [54] *categorizes the textual descriptions* of already solved reports and assigns developers to similar bugs. With a supervised Bayesian machine learning algorithm, this method correctly predicts 30 % of all developer-to-bug report mappings. An improved *semi-automated machine learning* approach [10, 11] works on open bug repositories and learns from a large number of already resolved reports the relationship between developers and bugs. It classifies new incoming reports with the help of text categorization and recommends a few developers that have worked on similar problems before. The evaluation reports a high precision between 57 % and 64 % for two out of three large open source projects. *Develect* [146] applies a similar approach, but it matches the lexical similarities between the vocabulary of bug reports and the diffs of developers' source code contributions. Its evaluation achieves 33.6 % precision for the most qualified developer and 71.0 % recall that the perfect match is within the top ten. As an alternative to comparing and classifying bug reports, the *developer selection* approach [41] studies the assignment of change requests in open source projects and finds out that source code repositories also provide valuable information for assigning change requests. With information retrieval techniques, the proposed method identifies a set of best candidates that have resolved similar change request in source code. An *extension to XFinder* [124] builds on the same idea; it identifies bug-related source code and then recommends proper expertise for changing these entities. A study [12] with project experts confirms that both source code and bug repository approaches are good at finding suitable developers.

As bug reports are still reassigned to better suited developers [94], novel approaches enlarge the analysis scope. *Bug tossing graphs* [115] improve other bug triage methods by revealing the relationship between reassignments, developers, and bug reports. Their results present a reduction in reassignments by up to 72 % and an improvement in accuracy of 76 % for the first five recommendations. Another *framework for automated bug triage* [26] further suggests to consider not only the bug history and software repositories but also the developer's expertise, workload, and personal preferences. Finally, *WhoseFault* [192] is similar to our developer ranking metric because it also considers anomalies in source code. Starting with spectrum-based fault localization to find suspicious source code entities, it mines history information for expertise and creates a weighted mapping between locations and developers. With a probability of 81 %, it produces the suitable developer in the first three recommendations. As WhoseFault and our approach have been developed independently and in parallel, there are also two major differences. First, we combine suspiciousness and confidence values with a harmonic mean and apply the more efficient Ochiai formula instead of Tarantula. Second, we focus on fast response times and so limit our analysis to methods instead of each single statement.

In contrast to our developer ranking metric and WhoseFault, previous approaches are generally applicable but their recommendation accuracy for a specific failure is limited. Other approaches consider either the entire system so that the search space is too large or they require similar failure reports which excludes new kinds of failures. Our metric restricts the search space to anomalies only. As anomalies are likely to include failure causes, we are able to provide more accurate developer rankings. In 80 % of all cases, the suitable expert is within the first five recommended developers. Although we require at least one failing test case, we think that often its implementation can be derived from bug reports without knowing failure causes.

8.2.4. Observation of Program Behavior

Due to the fact that our PathFinder is a back-in-time debugger, we start its related work with omniscient debuggers. Then we present simulated back-in-time debuggers that reverse execution or re-execute programs to access past events. Finally, we conclude with a broader overview of interesting debugging tools.

To follow infection chains from observable failures back to their root causes, omniscient debuggers record all executed events and present the collected data post-mortem. So, they allow developers to navigate an entire program history and answer questions about the cause of a particular state. The *omniscient debugger (ODB)* [135] records every event, object, and state change until execution is interrupted. However, the required dynamic analysis is quite time- and memory-consuming. The performance slows down up to 300 times and the memory consumes up to 100 MB per second. *Unstuck* [108] is the first back-in-time debugger for Smalltalk. As the tool stores execution traces in memory like ODB, it suffers from similar performance problems and relatively small traces. *WhyLine* [128] allows developers to ask a set of “why did” and “why didn’t” questions about the entire execution history. Combining static and dynamic slicing, call trees, and several other algorithms, the approach can answer, for example, why a line of code has not been reached. *Traceglasses* [188] records compact execution events of Java applications. By querying and transforming these events, developers have a comfortable navigation through large execution trees. *JHyde* [104] is a hybrid debugger for Java that integrates declarative with back-in-time debugging. With the help of generated hypotheses, developers can explicitly follow the infection chain backwards. Moreover, there are already straightforward back-in-time debuggers in commercial development environments such as Microsoft’s IntelliTrace for Visual Studio 2010.

However, all of these omniscient debuggers do not scale well with long execution traces including intensive data. For that reason, other approaches aim to circumvent these issues by focusing on performance improvements in return for a more complicated setup. The *trace-oriented debugger (TOD)* [175] combines an efficient instrumentation mechanism and a specialized distributed database for capturing exhaustive traces. This approach requires considerable infrastructure and set-up costs, and moreover imposes a high run-time overhead and resource requirements. Later, a novel indexing and querying technique [174] ensures scalability to arbitrarily large execution traces and offers an interactive debugging experience that outperforms existing back-in-time debuggers. *Object flow analysis* [141, 142] 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. Tracing is fast and memory consumption is low but it requires an adapted virtual machine and discarding events limit the approach because failure causes might be included in objects that are no longer in use. The *Compass debugger* [74] builds on top of the object flow analysis and presents an innovative user interface for back-in-time debuggers. Besides standard method call trees, this debugger allows developers to follow corrupted objects in order to find failure-inducing methods.

Instead of recording all events until the program stops, simulated back-in-time debuggers rely on reversing program execution. *ZStep95* [139] for the functional programming language Lisp allows developers to step a program in both forward and backward direction. In doing so, a call graph is visualized in real-time so that changes are directly visible. There are also *reverse debuggers for stack-based imperative programming languages* such as Java and C. In Java [50] new operational semantics that define reversed byte code instructions and side-effect logs which can be restored later on allow developers to execute programs backwards. In C [129] a virtual machine simulates the program execution in both directions and records run-time data depending on developers' needs. Although all approaches are able to go back in time, they are limited to step by step debugging. Developers cannot directly follow cause-effect chains because the entire execution history is missing.

Another kind of simulated back-in-time debugging is to periodically record complete program checkpoints and to re-execute them later on. During this partial re-execution, the program can be analyzed and stopped when required. With the break at an earlier point in time, developers have the impression to debug backwards. *Igor* [71] delegates the process of checkpointing to the operating system that then incrementally stores memory snapshots. With this expensive method, it allows reverse execution, selective searching of past data, and the substitution of program entities. *Spyder* [7] combines dynamic slicing and backtracking to identify causes and effects between statements and state. While automated dynamic slicing determines affected statements [8], backtracking restores program states from re-executed checkpoints at previously defined breakpoints. A more lightweight *replay debugger for Standard ML* [200] creates checkpoints as first-class continuations. This allows a flexible mechanism to replace execution and to provide reverse execution for the user and the interpreter. *Bidirectional debugging* [37] provides not only forward and backward commands by checkpointing, but also I/O-logging that ensures a deterministic re-execution. Previous approaches could not guarantee the same execution path which sometimes leads to wrong results. In general, so called record and replay techniques such as *DejaVu* [48] prevent this problem by first recording non-deterministic program points and later replaying their results. *Jockey* [187] is a user-space library that combines record and replay with checkpoints. It rewrites all non-deterministic system calls and CPU instructions and takes periodic snapshots to enable time traveling in distributed Linux programs. *Backstep* [43] inserts an undo command into the debugging process of the Java Eclipse development environment. Each time a method is entered, a checkpoint is created that allows for restarting the method from its invocation. However, all checkpoint approaches have scalability issues because they strongly depend on the size and frequency of their checkpoints.

Apart from back-in-time debuggers, there are other debugging tools that support the search for failure causes. *Coca* [61] is a debugger for C that automatically adds breakpoints at control flow and data events. The analysis is done on the fly and the program stops as soon as a specified event occurs. *Query-based debugging* [133] allows an efficient search for relationships in large object spaces after a program stops. In a more dynamic version [134], similar queries continuously check objects while running and instantly stop if a violation is found. Later on, *snapshot query-based debugging* [173] further optimizes performance and proposes a more specialized query language. *Pervasive debuggers* [107] allow developers to inspect concurrent and distributed applications. In a virtualized environment, such debuggers have full control over each component and their interactions. At each point in time, they can stop them in a consistent state independent of network latency, different programming languages, and multiple hosts. *Object-centric debugging* [181] proposes to primarily consider objects and their interactions instead of stack-based run-time environments. The approach sees objects as the key abstraction and presents a new debugger that answers more object-related questions. The *visual symbolic debugger* [96] shows all possible execution paths without running the program. The applied symbolic execution helps developers to understand behavior in small parts of the code. Moreover, there are visualization concepts that directly support debugging. The *data display debugger* [221] shows data structures as graphs that can be refined step by step. The *debugging canvas* [57] offers a two-dimensional pan-and-zoom surface for arranging code bubbles [39] that represent call paths, variable values, and source code snippets. So, developers see everything regarding the current debugging task at a glance.

Compared to the presented debugging tools, our PathFinder is a lightweight and specialized back-in-time debugger for localizing failure causes in unit tests. Our approach quickly provides established dynamic views due to our concept of step-wise run-time analysis. We do not record each event beforehand; rather developers specify interest in particular parts which are then refined on demand in additional test case re-executions. This ensures fast response times and low memory consumption because we only record requested data. As test cases are reproducible by definition, we do not need additional checkpoints or record and replay techniques. During debugging, our PathFinder includes the important characteristic of immediacy, which most of the related debuggers are missing [201]. Furthermore, our behavior navigation and its classified traces allow developers to select erroneous behavior directly. Without this concept, developers require more internal knowledge to isolate the infection chain and to decide which path to follow. Regarding the other presented debugging approaches, we think that they are valuable concepts for future work. For example, a query language for searching in object spaces could further help to explore execution histories.

8.2.5. Isolation of Corrupted State

Our state navigation combines design by contract with automatically derived invariants in order to isolate corrupted state and highlight infection chains. Apart from these related topics, we further discuss alternative approaches for harvesting type invariants.

The roots of *design by contract* can be traced back to the work of Floyd [75]. In addition to the original design by contract implementation in Eiffel [151], we can find numerous extensions in other programming languages such as C++ [92], Java [130], Python [170], and Smalltalk [42]. While all of them support the classic design by contract that checks pre- and post-conditions of methods and program invariants, only a few completely support recursive assertion checks and the old statement which allows access to preceding method states. Furthermore, they differ in their implementation strategies (annotations, inheritance, or aspect-oriented programming [126]), granularity level (methods, classes, and components), and contract scopes (grouping and run-time activation) [106].

An alternative to manually specifying contracts is to automatically derive invariants from correct program behavior. *Daikon* [70, 68] comprises a set of dynamic techniques for inferring generalized program state from execution traces. It observes occurring objects and summarizes their specific properties into invariants such as non-zero properties and containment relationships. With such an inductive analysis, the approach reveals the implicit run-time information necessary for creating contracts. These invariants detect not only corrupted state, but also help in generating test cases and repairing inconsistent data structures. However, the first version suffers from a scalability issue which creates too many results, misses important invariants, and requires too much time. Later, an *extension to Daikon* [69] solves this problem by optimizing polymorphism and filtering unchanged values. *Carrot* [176] experiments with a subset of Daikon's invariants and tries to localize failure causes. However, the experimental results are unsatisfactory. *Diduce* [100] extends Daikon's approach and derives invariants on the fly. During program execution, it monitors its run-time and gradually relaxes invariants of observed objects. After a while, developers receive bug reports only for corner cases, which then identify failure causes quickly and automatically. *Screeners* [1] further optimize the run-time overhead and decrease it to only 14 %. *ClearView* [164] also learns and monitors invariants on the fly but it also applies this knowledge to automatically patch upcoming failures.

As soon as a violation occurs, it generates candidates to hold the invariant and continues the application with changed state or control flow. Finally, a comparative study [172] of programmer-written and automatically inferred contracts concludes that a combination of both methods is most promising. While manual contracts have a high quality but fewer results, Daikon's contracts have many results but less quality.

Existing approaches for discovering invariants ignore type information because they are implemented in statically typed programming languages. Our type harvester is mostly related to approaches that allow for obtaining type information in dynamically typed languages. Consequently, there are two main categories to consider: type inference and run-time type collection. *Type inference* for dynamically typed programming languages has been researched early on [198]. More recent approaches [4, 5, 38, 81, 171, 195] mostly focus on obtaining type information for interfaces, members and local variables. As they only rely on static analysis, the precision and recall scores are often low. *Run-time type collection* is scarcely used in a way that allows programmers to exploit its results. *Type feedback* [109] is a virtual machine technique that drives just-in-time compiler optimization decisions based on dynamically collected type information. This information is confined to the virtual machine. *Profile-guided typing* [80] augments type inference for the Ruby programming language. This approach requires the code under observation to be instrumented—unlike type harvesting, which leaves code untouched. The development environment extension *Hermion* [183] for Squeak and its successor *Senseo* [184] for Eclipse are designed to provide developers with additional run-time information during static source code navigation. Both extensions, among other things, improve IDE usability by exploiting dynamic type information, effectively resolving navigation issues related to late-binding. For example, Hermion restricts sender and implementor lists in Smalltalk to relevant run-time types only. To that end, it permanently collects method signatures and receiver types during normal program use. Conversely, type harvesting can be executed on demand with test suites and harvested information is more extensive.

Our state navigation combines design by contract with the dynamic discovery of invariants and reveals infection chains by mapping corrupted state onto execution traces. Our contract system in Smalltalk is similar to our previous PyDCL implementation for Python [106]. It supports the classic design by contract including recursive assertions and the old statement. Furthermore, it implements contract grouping and activation at run-time. These contracts are then automatically generated with invariants by our inductive analysis. We harvest run-time data from the hidden test knowledge analogous to Daikon with the difference that developers can adapt the analysis to their needs. They decide about object properties (types, value ranges), scope (program entities), and accuracy (performance optimizations). Independent of their decision, they can incrementally refine and update the derived invariants in additional test runs. Finally, we map violated contracts on execution histories of failing tests in order to highlight infection chains. Thus, these state anomalies further support developers in understanding and localizing failure causes.

8.3. Software Visualization for Debugging

In general, software visualization summarizes and represents information about software systems in order to enhance the comprehension of programs. This definition also includes debugging activities because developers obtain a clearer understanding of the problem [19]. However, as software visualization includes a broad range of techniques [59], we limit our discussion to closely related approaches. We briefly consider tree maps for representing the static structure of a system and the visualization of execution traces as a technique for illustrating program behavior.

PathMap represents the results of our structure navigation in form of a *tree map* [119]. This visualization summarizes arbitrary hierarchical structures by recursively subdividing a given area into smaller rectangles. Since then, several other approaches have improved the standard layout algorithm: they *prevent long rectangles* that are difficult to see [194], *cascade rectangles* to obtain a depth effect [145], or highlight objects that are also *neighbors in the hierarchical structure* [24, 207]. Apart from tree maps, there are also alternative approaches for compact layouts of hierarchical data structures [21, 132] Nevertheless, we chose a tree map layout because of its level of awareness, low space requirement, and straightforward implementation. We base our tree map on the standard layout and limit the hierarchy depth to four, make large rectangles zoomable on demand, and draw classes with a thicker border to emphasize their included methods.

There are plenty approaches to software visualization that present behavior using execution traces [53, 99]. One of the first [116] presents several prototypes ranging from a *global overview of large traces to lowest system level events*. Subsequent approaches primarily focus on presenting large amounts of trace data. The *execution pattern view* [56] automatically classifies repetitive behavior into high order execution patterns. The *Vizz-Analyzer* [144] combines static and dynamic analysis to generate static call graphs with run-time information. *High-level polymetric views* [62] are a lightweight approach for visualizing condensed dynamic data with measurements that focus on the understanding of certain aspects such as objects lifetime or the communication architecture. The *call graph analyzer* [35, 36] visualizes execution traces in a 2.5D environment and extends this information by a number of other analysis approaches. Also, reverse engineered *sequence diagrams* are very common but with the fact that they do not scale [30]. To solve this issue, *circular bundle views* [51] present a scalable visualization inside a circle. Although all approaches are useful in their specific scenario, no approach can be generalized for various tasks in program comprehension and debugging [161]. Most such visualizations are rarely used during software development [183]. One reason might be the missing integration into development environments or the neglected aspect of low setup and performance cost that our Path Tools addresses [44].

8.4. Dynamic Analysis for Recording Execution Traces

Dynamic analysis considers the behavior and properties of a running program [20]. This information is fundamental for several debugging activities because it provides developers insights into corrupted program behavior. The most common dynamic analysis techniques collect run-time information by instrumenting source code with additional log code, extending the virtual machine, or running the system under control of a debugger [99]. Each technique has its own assets and drawbacks. However, all of them slow down the program execution significantly, produce a large amount of fine-granular trace data, and are difficult to handle because of missing transparency for developers. For that reason, several approaches suggest optimizations with respect to the decrease of run-time overhead and the compression of collected data. *Efficient path profiling* [22] proposes a fast algorithm to select profile information and compress its trace data. It determines the number of executed acyclic paths in a method and subsumes common basic block and edge profilings. *Encoded program executions* [179] select subsets of traced information and then compact it by inferring and encoding its structure. *Dynamic instrumentation* [153] inserts and removes log code on the fly and so keeps performance overhead low.

Programming languages supporting meta-programming features allow further dynamic analysis approaches with a high degree of flexibility. *Behavioral reflection* [58] offers a behavioral middle layer that abstracts from the actual implementation details and provides a standard API for all tools to use. This layer considers the run-time system as a collection of reified first-class entities on a meta-level. Meta objects implement a transparent tracing mechanism that captures and stores run-time behavior. This concept allows for specifying the kind and amount of dynamic information precisely and it provides a unified access to run-time information [183]. However, since behavioral reflection is implemented by an extensive use of meta programming, the performance issue is still open. The *Spy framework* [31] inserts dedicated code before or after method executions via method wrappers [40]. It is flexible enough to easily build several tools for profiling, test coverage, and type inference. To extend programming languages without access to the meta-level, developers can rely on aspect-oriented programming [126] or emulation engines. *Aspects* separate the tracing mechanism from the source code and allow developers to select an adequate level of detail [91]. The performance of this approach scales very well since the extended source code can be optimized by the run-time environment. Instead of executing machine code directly, *Nirvana* emulates it and send callbacks to its *iDNA* tracing engine [34]. This approach has only a low run-time overhead and further allows the compression and selection of trace data in an external dynamic analysis tool.

Compared to our incremental dynamic analysis and the corresponding Path dynamic analysis framework, we rely on method wrappers and the meta-programming features of our host language Smalltalk. We have a flexible technique like the Spy framework that allows developers to choose from and implement new dynamic analysis approaches. However, our *incremental* and *interactive* properties ensure a low performance overhead for single runs and recorded traces include only data being relevant for developers.

8.5. Summary

We presented related work to our test-driven fault navigation, incremental dynamic analysis, and Path Tools framework. Starting with testing in general, we discussed approaches in particular that consider tests as examples and analyze their hidden knowledge. After that, we introduced the corresponding research areas for our debugging approach and each of its navigation steps. Scientific debugging establishes a relation to our debugging process. Spectrum-based fault localization lays the foundation for our structure navigation. Approaches which determine developer expertise relate to our team navigation and its developer ranking metric. The observation of program behavior is important to our behavior navigation and its back-in-time debugging. The detection of invariants to identify corrupted state is similar to our state navigation. Furthermore, we looked at visualized execution traces and dynamic analysis techniques that are not directly related to debugging concepts.

9 Conclusion

In this chapter, we summarize our work and present interesting directions for future work. First, we sum up our contributions and how these answer our research question (Section 9.1). Second, we conclude this thesis by presenting three additional projects that already extend our Path Tools framework in order to support debugging even better (Section 9.2).

9.1. Summary

In this dissertation, we deal with the problem of debugging reproducible failures. During debugging, developers try to understand what causes observable failures by following infection chains back to their root causes. In doing so, they primarily have to rely on their intuition because contemporary debugging methods and tools are limited with respect to investigating such infection chains. This often leads to disorganized trial and error approaches which make debugging a hard and time-consuming activity. For these reasons, experienced developers mostly rely on the scientific method and its hypothesis-testing to systematically narrow down root causes. Despite this method debugging is still a tedious task. Developers require much knowledge about the program to create and evaluate proper hypotheses and, moreover, contemporary debugging tools provide no or only partial support for this systematic hypothesis-testing. With respect to these debugging problems, we summarize our research question as follows: "How can we efficiently support developers in creating, evaluating, and refining failure cause hypotheses so that we reduce time and effort required for debugging?"

Methodic contribution

> Our *test-driven fault navigation* is a debugging guide that integrates anomaly detection into a breadth-first search for systematically creating, evaluating, and refining failure cause hypotheses.

We offer four specific navigation techniques that systematically lead developers to failure causes based on new perspectives of test cases and the scientific method. We consider test cases not only as a way to verify if a failure occurs or not but also as reproducible entry points and a valuable source of hidden knowledge. While the first perspective offers deterministic and exemplary behavioral paths through the system, the latter reveals

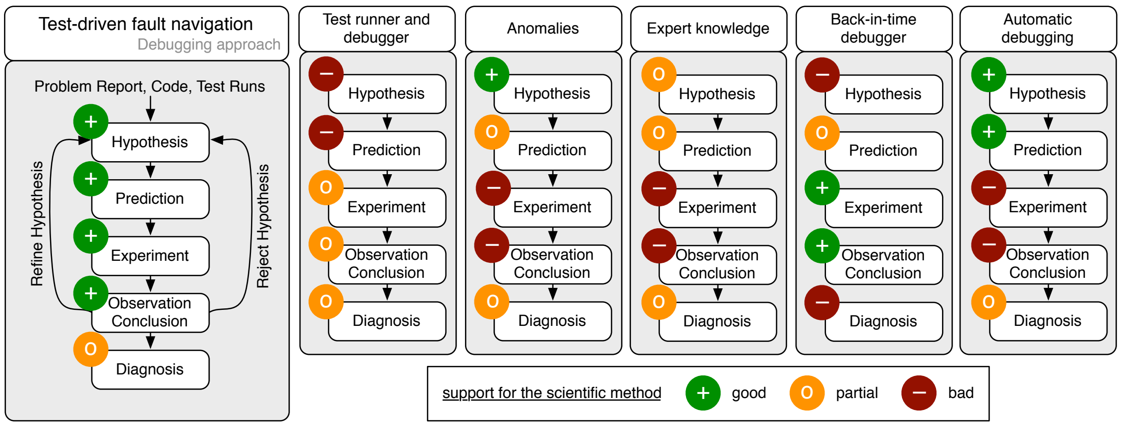


Figure 9.1.: The final comparison between our test-driven fault navigation and other contemporary debugging approaches reveals that our approach comprehensively supports the scientific method.

anomalies by comparing passing and failing test cases. Our test-driven fault navigation utilizes these two test case perspectives to follow complete infection chains backwards and to integrate anomalies for guidance through the large amount of run-time data. We propose four specific navigation techniques that together realize the scientific method in form of a breadth-first search. *Structure navigation* localizes suspicious system parts and restricts the initial search space with the help of spectrum-based anomalies. Developers obtain several starting points that are likely to include failure causes. *Team navigation* recommends experienced developers for helping with failure causes. We limit the search to suspicious system parts and so propose suitable experts even if root causes are still unknown. *Behavior navigation* allows developers to follow the infection chain of failing test cases back in time. With the help of anomalies, we further classify the entire execution history and so navigate developers through the large amount of run-time data. *State navigation* automatically uncovers parts of the infection chain by comparing used objects of passing and failing test cases. We harvest common object properties, create dynamic contracts, and reveal anomalies that support developers in understanding corrupted state. All techniques together form a coherent guide that integrates anomalies to systematically navigate developers from the occurrence of failures back to their root causes.

With respect to the scientific method, Figure 9.1 compares our approach with the state of the art in debugging methods. Since we combine the advantages of several techniques such as spectrum-based fault localization [122], omniscient debugging [135], and likely invariants [70], we are able to provide a comprehensive method for the systematic testing of hypotheses. We support the creation and refinement of *hypotheses* with our structure navigation that reveals anomalies and so restricts the search space for possible failure causes. As the *prediction* requires thorough comprehension about suspicious program entities, our team navigation automatically recommends suitable developers that are likely able to explain the expected behavior. To *experiment* with entire infection chains, our behavior navigation allows developers to access classified execution histories with all their object states. We contribute to the *observation and conclusion* with our behavior and state navigation which automatically reveal anomalies in infection chains and so highlight possible failure causes. The *diagnosis* is only partially supported because our approach still requires manual decisions about real failure causes. In summary, our test-driven fault navigation supports an entire breadth-first search for debugging reproducible failures by systematically leading developers along failure causes on infection chains.

Technical contribution

> Our *incremental dynamic analysis* ensures the efficiency of test-driven fault navigation by interactively splitting expensive analyses over multiple test runs and so creating an experience of immediacy when debugging with our approach.

We collect the required run-time data for debugging with our approach on demand. Based on the idea of test cases as reproducible entry points into behavior, we distribute the dynamic analysis across multiple test runs depending on developers' needs. Starting with an initial overview of run-time data, developers interactively refine their understanding step by step. Each refinement step only collects the requested information during additional test runs. This approach reduces the effort for providing an initial overview and optional refinements impose just a minimum of additional cost. So, we can ensure fast response times when debugging with our approach as run-time data is only collected when needed.

For our four test-driven fault navigation techniques, we propose three specific incremental dynamic analysis techniques. *Refined coverage analysis* provides fast access to method coverage and on-demand refinements at statement level. Our structure and team navigation require this run-time data for computing spectrum-based anomalies at different levels of detail. *Step-wise run-time analysis* collects the execution history of a specific test case for our behavior navigation. It begins with a shallow analysis of the call tree and allows developers to refine specific details later on. *Inductive analysis* harvests common object properties such as run-time types and value ranges of passing test cases. In doing so, developers choose the level of detail and kind of information to be collected for the dynamic contracts of our state navigation. With our three incremental dynamic analysis techniques, we ensure immediate access to run-time data and so encourage the frequent use of our test-driven fault navigation approach and its corresponding debugging tools.

Implementation contribution

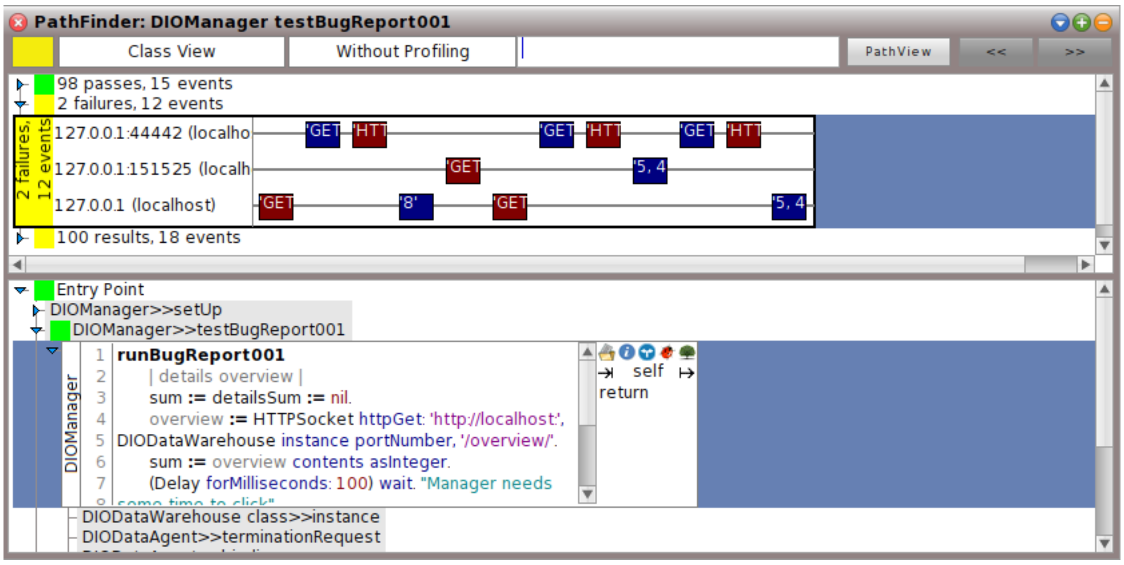
> Our *Path Tools framework* implements test-driven fault navigation based on our incremental dynamic analysis for the Squeak development environment. With the help of this implementation, we have successfully applied our approach to several Smalltalk projects.

We implement both test-driven fault navigation and incremental dynamic analysis in our Path Tools framework for the Squeak/Smalltalk development environment. It consists of three tools and one analysis framework that together realize all four navigation techniques, provide fast access to run-time data, and reveal the hidden test knowledge. *PathMap* is an extended test runner for our structure, team, and state navigation. With the help of our refined coverage analysis, it collects spectrum-based anomalies and presents them in a compact system overview. In addition to this, PathMap recommends experienced developers and harvests common object properties from passing test cases. *PathFinder* is a lightweight back-in-time debugger for our behavior and state navigation. It builds on top of our step-wise run-time analysis in order to allow developers interactive and immediate access to the execution history of a specific test case. *PathBrowser* extends Smalltalk's standard source code editor to benefit from the hidden test knowledge; it shows the test coverage results of our structure navigation and represents the dynamic contracts of our state navigation. All tools are based on our flexible analysis framework that offers several hooks for implementing our incremental dynamic analysis. With the help of our Path Tools framework, developers can localize suspicious system parts, learn about other developers for help, and debug emphasized infection chains back to their failure-inducing origins.

The evaluation demonstrates that our test-driven fault navigation is practical for bringing developers closer and faster to defects. We have successfully applied our approach and the corresponding Path Tools framework to several Smalltalk projects and can conclude that developers require less time and effort for debugging. In particular, our structure navigation limits the initial search space to a large extent; our team navigation possesses a high accuracy in recommending qualified developers; the emphasized infection chains of our behavior navigation assist developers in following failure causes easily; our state navigation evenly highlightes corrupted state along infection chains. While debugging with our Path Tools, our incremental dynamic analysis ensures short response times and fast access to the necessary run-time data. All in all, we conclude with regard to our research question—our test-driven fault navigation efficiently supports developers in creating, evaluating, and refining failure cause hypotheses and reduces debugging cost with respect to time and effort.

9.2. Outlook

In addition to our presented test-driven fault navigation for debugging reproducible failures, we provide three directions for future work. First, we will improve our approach for debugging non-deterministic failures in network environments and multi-threaded applications. Second, we will extend our step-wise run-time analysis with a rich query language that allows developers to answer "why does questions" about corrupted state. Third, we will support the correction of defects by highlighting anomalies in novel and compact source code views. With the help of several student projects, we have already



```
runBugReport001
    | details overview |
    sum := detailsSum := nil.
    overview := HTTPSocket httpGet: 'http://localhost:',
DIODataWarehouse instance portNumber, '/overview/'.
    sum := overview contents asInteger.
    (Delay forMilliseconds: 100) wait. "Manager needs
some time to click"
```

Figure 9.2.: Replay-driven fault navigation extends our approach for debugging distributed failures.

implemented our ideas in prototypes that we describe in the following:

Replay-driven Fault Navigation widens our approach for debugging non-deterministic failures in networks and multi-threaded applications [72, 73]. As non-deterministic failures are not reliably reproducible, they are hard to debug because developers cannot systematically test their hypotheses and follow failure causes back to defects. For example, in distributed Web applications with their custom and timing-dependent communication interfaces, non-deterministic failures occur frequently and present developers with a challenge [15, 156]. To localize failure causes in distributed Web applications, we propose a lightweight *record and refine* extension to our approach. It allows developers to reproduce non-deterministic failures, reveal anomalies in their communication, and refine run-time behavior in specific service implementations[1].

To debug non-deterministic failures, our approach consists of three steps. In the initial record phase, we log communication of non-deterministic failing test cases and divide their schedules into failing and successful runs. After that, we analyze the differences between schedules to detect anomalies that are likely to cause the non-deterministic failure. Finally, we constrain the system to a failing schedule by modifying it with a traffic shaper. This tool removes timing-dependent communication as a source of non-determinism and always replays the same failing pattern. If the constrained system reproduces the failure, we are able to apply our step-wise run-time analysis and access more refined run-time

[1]Even if this work only deals with network communication, we argue that our approach can easily be adapted to debug multi-threaded applications. Services as well as processes are independent program entities that communicate with each other either distributed or on the same machine.

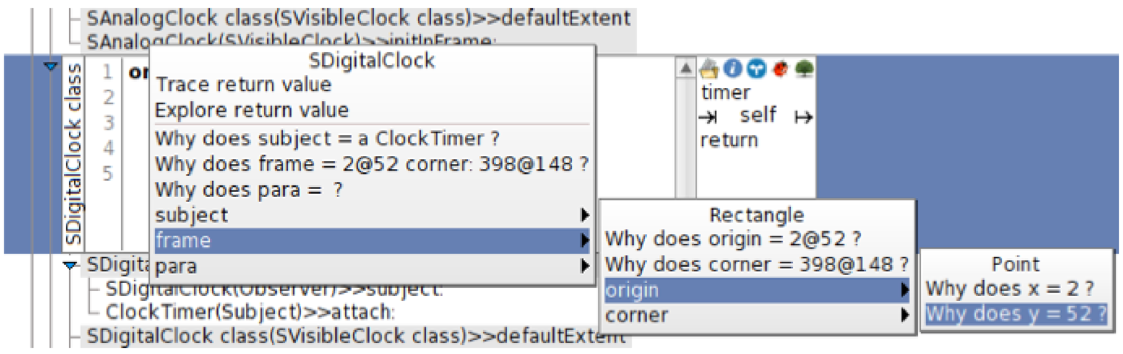

SAnalogClock class(SVisibleClock class)>>defaultExtent
SDigitalClock class
SDigitalClock
Trace return value
Explore return value
Why does subject = a ClockTimer ?
Why does frame = 2@52 corner: 398@148 ?
Why does para = ?
subject
frame
para
ClockTimer(Subject)>>attach:
SDigitalClock class(SVisibleClock class)>>defaultExtent
timer
self
return
Rectangle
Why does origin = 2@52 ?
Why does corner = 398@148 ?
origin
corner
Point
Why does x = 2 ?
Why does y = 52 ?

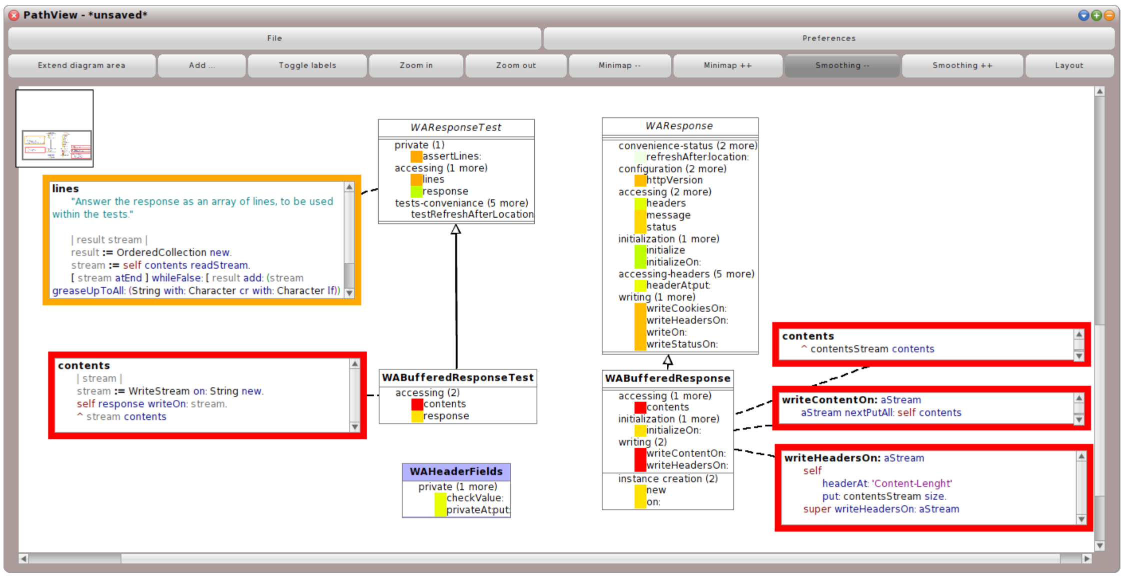
Figure 9.3.: PathFinder-Y is an extension for our lightweight back-in-time debugger that allows developers to query the execution trace and ask why-questions about failure causes.

data. In doing so, we replay the recorded, now reproducible failing test case to collect events between services and execution histories of specific service implementations on demand.

Figure 9.2 shows our PathFinder extension to support back-in-time debugging of non-deterministic communications. In the upper part, we render the input and output events between services and highlight anomalies in their communication schedules. In the lower part, developers can refine specific messages and see what happens inside the implementation that processes this request. PathFinder presents the corresponding call tree and allows developers to refine run-time data with our incremental dynamic analysis.

PathFinder-Y extends our lightweight back-in-time debugger with a query engine that allows developers to ask "why"-questions about failure causes. Our query language simplifies the tracking of corrupted state by answering *Why does an object have a specific value?* We guide developers to the corresponding method in the execution that generates this specific value. With the help of so called object traces, developers can comfortably follow infected objects back to their root causes.

The query language is similar to the Whyline approach [128] that allows developers to answer all kinds of why-questions. Although this approach supports developers in localizing failure causes, the required performance overhead for a full dynamic analysis limits its practicality. To solve this drawback, we combine why-questions with our incremental dynamic analysis approach. We extend the specific step-wise run-time analysis with object traces and implement a lightweight whyline debugger called PathFinder-Y. Our extension collects all object IDs in the entire execution history and links them to their corresponding call nodes. So, we know what method calls influence which objects. This analysis must be done only for the first query and is much faster than creating deep copies of all objects. If developers are interested in the origin of a specific state, we collect the corresponding object with our refinement analysis, remember its ID, and generate related why-questions. While developers choose a specific question, we already



```
lines
	"Answer the response as an array of lines, to be used
within the tests."

	| result stream |
	result := OrderedCollection new.
	stream := self contents readStream.
	[ stream atEnd ] whileFalse: [ result add: (stream
greaseUpToAll: (String with: Character cr with: Character lf))
```

```
contents
	| stream |
	stream := WriteStream on: String new.
	self response writeOn: stream.
	^ stream contents
```

```
contents
	^ contentsStream contents
```

```
writeContentOn: aStream
	aStream nextPutAll: self contents
```

```
writeHeadersOn: aStream
	self
		headerAt: 'Content-Lenght'
		put: contentsStream size.
	super writeHeadersOn: aStream
```

Figure 9.4.: PathView is a model-based source code editor that extracts views from infection chains to assist developers in correcting defects.

refine all objects with the same ID in our call tree so that we are able to create an object trace along the execution history. In doing so, we have recorded enough information to answer the specific question by comparing state changes of the original object with all other traced objects. If a state change answers the question, we guide developers to the related method in the execution. Due to the fact that we do not need to collect all objects beforehand but rather access the required run-time information incrementally, we are able to answer why-questions in a short amount of time.

Figure 9.3 illustrates our prototype and the generated questions. For example, there is a why-question about the rectangle scope of the frame object and the y-coordinate of its origin point. The answer to this question leads to the executed method that initializes or changes this value to 52. In this first prototype, we rely on the comparison of object strings before and after a method execution. Therefore, our approach works for all classes that implement Smalltalk's `printString` method. Among others, this includes all primitive types such as booleans, numbers, and collections.

PathView supports developers in fixing root causes by extracting related program entities from infection chains and highlighting relationships between anomalies. Based on the execution history of a failing test case, we automatically generate a UML-like view of all involved classes, methods, and their relationships. In addition to this, we highlight spectrum-based and state anomalies to assist developers in understanding corresponding methods and source code. PathView complements our Path Tools framework as it provides a mid-level abstraction for suspicious program entities between the entire system overview of PathMap and the detailed source code within PathBrowser. Thus, developers are able to see at a glance how suspicious methods are related to each other, the system structure, and the defect. We argue that such views help developers not only to correct defects but also to prevent the introduction of new and similar failures.

The UML-like representation of our PathView editor includes a single-source and round-trip engineering approach. We only consider two primary artifacts as the foundation for all models. First, *source code* is the executable specification of our program and the abstract description of the system's structure. With the help of static analysis, we can derive classes, methods, and their relationships. Second, *test cases* reveal behavioral examples and yield a hidden source of information. While failing test case examples serve as input for generating failure-related views, the hidden test knowledge provides associations with harvested type information. All further information for a model-based visualization can be derived from these artifacts. To complete the round-trip, developers can interactively change and extend the visualization. If they change something in the model, they automatically change the source code. For example, the replacement of an inheritance relationship immediately leads to a new super class of the changed class in source code. Also changes in source code instantly update related views. There are only a few conflicts that cannot be solved automatically. We mark these problems so that developers directly see which information is outdated. In addition to these features, we also offer storing and merging of views in source code management systems. With this approach, we blur the boundaries between separate models and the source code base. Thus, we bring together the benefits of models with their overview and source code as executable specifications without the drawback of outdated artifacts due to necessary synchronization issues.

Figure 9.4 presents PathView for our Seaside typing error. Test cases on the left and response classes on the right represent their relationships and suspicious methods. Each class only shows methods that have been called during the failing test case execution. All other methods are hidden because they are irrelevant for fixing the defect and would unnecessarily clutter the visualization. We mark anomalies with small boxes before the method name and open the most suspicious methods with their corresponding source code. These source code entities possess a high probability to have a close relation to failure causes and the defect.

Closing Remarks

In this thesis, we presented test-driven fault navigation as a debugging guide that integrates anomaly detection into a breadth-first search for systematically creating, evaluating, and refining failure cause hypotheses. Based on the scientific method, we combine spectrum-based fault localization, developer recommendation, back-in-time debugging, and likely invariants to further support developers in following infection chains back to their root causes.

However, almost all novel approaches have a few shortcomings that prevent them from replacing outdated but still prevalent debugging tools. We argue that our test-driven fault navigation solves some of these open issues and so represents a step towards the applicability of debugging research in practice.

September 25, 2013
Michael Perscheid

List of Figures

List of Tables

Publications

Journal Publications

Michael Perscheid, Michael Haupt, Robert Hirschfeld, and Hidehiko Masuhara. Test-driven Fault Navigation for Debugging Reproducible Failures. In *Journal of the Japan Society for Software Science and Technology (JSSST) on Computer Software*, 29(3):188–211, 2012.

Conference Publications

Michael Perscheid, Damien Cassou, and Robert Hirschfeld. Test Quality Feedback - Improving Effectivity and Efficiency of Unit Testing. In *Proceedings of the Conference on Creating, Connecting and Collaborating through Computing*, C5, pages 60–67. IEEE, 2012.

Lauritz Thamsen, Anton Gulenko, Michael Perscheid, Robert Krahn, Robert Hirschfeld, and David A. Thomas. Orca: A Single-language Web Framework for Collaborative Development. In *Proceedings of the Conference on Creating, Connecting and Collaborating through Computing*, C5, pages 45–52. IEEE, 2012.

Robert Hirschfeld, Michael Perscheid, and Michael Haupt. Explicit Use-case Representation in Object-oriented Programming Languages. In *Proceedings of the Dynamic Languages Symposium*, DLS, pages 51–60. ACM, 2011.

Michael Perscheid, Michael Haupt, Robert Hirschfeld, and Hidehiko 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.

Michael Haupt, Michael Perscheid, and Robert 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.

Michael Perscheid, Bastian Steinert, Robert Hirschfeld, Felix Geller, and Michael 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.

Michael Haupt, Michael Perscheid, Robert Hirschfeld, Lysann Kessler, Thomas Klingbeil, Stephanie Platz, Frank Schlegel, and Philipp Tessenow. PhidgetLab - Crossing the Border from Virtual to Real-World Objects. In *Proceedings of the Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE, pages 73–77. ACM, 2010.

Robert Hirschfeld, Michael Perscheid, Christian Schubert, and Malte Appeltauer. Dynamic Contract Layers. In *Proceedings of the Symposium on Applied Computing*, SAC, pages 2169–2175. ACM, 2010.

Bastian Steinert, Michael Perscheid, Martin Beck, Jens Lincke, and Robert 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.

Workshop Publications

Tim Felgentreff, Michael Perscheid, and Robert Hirschfeld. Constraining Timing-dependent Communication for Debugging Non-deterministic Failures. In *Proceedings of the Workshop on Academic Software Development Tools and Techniques*, WASDeTT, accepted. Online, 2013.

Michael Perscheid. Dynamic Service Analysis: Test-driven Fault Navigation for Debugging Reproducible Failures. In *Proceedings of the Joint Workshop of the German Research Training Groups in Computer Science*, page 216. m verlag mainz, 2012.

Michael Perscheid. Dynamic Service Analysis: Test-Driven Views for Enhancing Software Maintenance. In *Proceedings of the Joint Workshop of the German Research Training Groups in Computer Science*, page 210. m verlag mainz, 2011.

Michael Perscheid. Dynamic Service Analysis. In *Proceedings of the Joint Workshop of the German Research Training Groups in Computer Science*, pages 204–205. m verlag mainz, 2010.

Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. A Comparison of Context-Oriented Programming Languages. In *In Proceedings of the International Workshop on Context-oriented Programming*, COP, pages 1–6. ACM, 2009.

Books

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.

Technical Reports

Michael Perscheid. Demonstrating Test-driven Fault Navigation. In *Proceedings of the Fall Workshop of the HPI Research School on Service-Oriented Systems Engineering*, pages 133–143. Hasso-Plattner-Institut, 2012.

Michael Perscheid. Test-driven Perspectives for Debugging Reproducible Failures. In *Proceedings of the Fall Workshop of the HPI Research School on Service-Oriented Systems Engineering*, to appear. Hasso-Plattner-Institut, 2011.

Michael Perscheid. Understanding Service Implementations Through Behavioral Examples. In *Proceedings of the Fall Workshop of the HPI Research School on Service-Oriented Systems Engineering*, pages 89–100. Hasso-Plattner-Institut, 2010.

Michael Perscheid. Requirements Traceability in Service-oriented Computing. In *Proceedings of the Fall Workshop of the HPI Research School on Service-Oriented Systems Engineering*, pages 15-1–15-11. Hasso-Plattner-Institut, 2009.

Michael Perscheid. Eclipse Plug-ins. In *Proceedings of SAP Web Application Server Technology.* Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam, 2006.

Demonstrations

Michael Perscheid. Test-driven Fault Navigation for Debugging Reproducible Failures, *Symposium on Future Trends in Service-Oriented Computing*, June 2012.

Michael Perscheid and Robert Hirschfeld. Evaluating Test-driven Fault Navigation, *Empirical Evaluation of Software Composition Techniques Workshop (ESCOT) at MODULARITY: aosd 2012 conference*, March 2012.

Robert Hirschfeld, Michael Haupt, Peter Osburg, Michael Perscheid, Martin Beck, Stefan Berger, Gregor Gabrysiak, Thomas Kowark, Dominic Letz, David Tibbe, and Matthias Wagner. Tours and Traps—Complex Software, with Simple Tools, in Time, *European Conference on Object-oriented Programming (ECOOP)*, August 2007.

Other

Michael Perscheid. Debugging mit der test-getriebenen Fehlernavigation, *HPI Magazin*, February 2013.

Bibliography

[1] R. Abreu, A. González, P. Zoeteweij, and A. van Gemund. Automatic Software Fault Localization Using Generic Program Invariants. In *Proceedings of the Symposium on Applied Computing*, SAC, pages 712–717. ACM, 2008.

[2] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. van Gemund. A Practical Evaluation of Spectrum-based Fault Localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.

[3] W. Achtert, T. Becker, H. Biskup, D. Hellebrand, J. Herczeg, S. Krause, M. Kuhrmann, F. Maar, F. Marschall, M. Minich, F. Simon, and S. Ziegler. Industrielle Softwareentwicklung — Leitfaden und Orientierungshilfe. Technical report, BITKOM, 2010.

[4] O. Agesen, J. Palsberg, and M. I. Schwartzbach. Type Inference of SELF: Analysis of Objects with Dynamic and Multiple Inheritance. *Software: Practice and Experience*, 25(9):975–995, 1995.

[5] O. Agesen and D. Ungar. Sifting out the Gold: Delivering Compact Applications from an Exploratory Object-oriented Programming Environment. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA, pages 355–370. ACM, 1994.

[6] H. Agrawal, J. Alberi, J. Horgan, J. Li, S. London, E. Wong, S. Ghosh, and N. Wilde. Mining System Tests to Aid Software Maintenance. *Computer*, 31(7):64–73, 1998.

[7] H. Agrawal, R.A. Demillo, and E.H. Spafford. Debugging with Dynamic Slicing and Backtracking. *Software: Practice and Experience*, 23(6):589–616, 1993.

[8] H. Agrawal and J. Horgan. Dynamic Program Slicing. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI, pages 246–256. ACM, 1990.

[9] H. Agrawal, J. Horgan, S. London, and E. Wong. Fault Localization using Execution Slices and Dataflow Tests. In *Proceedings of the International Symposium on Software Reliability Engineering*, ISSRE, pages 143–151. IEEE, 1995.

[10] J. Anvik. Automating Bug Report Assignment. In *Proceedings of the International*

Conference on Software Engineering, ICSE, pages 937–940. ACM/IEEE, 2006.

[11] J. Anvik, L. Hiew, and G. Murphy. Who Should Fix this Bug? In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 361–370. ACM/IEEE, 2006.

[12] J. Anvik and G. Murphy. Determining Implementation Expertise from Bug Reports. In *Proceedings of the International Workshop on Mining Software Repositories*, MSR, pages 2–9. IEEE Computer Society, 2007.

[13] V. Araya. Test Blueprint: An Effective Visual Support for Test Coverage. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 1140–1142. ACM/IEEE, 2011.

[14] E. Arisholm. Dynamic Coupling Measures for Object-Oriented Software. *Transactions on Software Engineering*, 30(8):491–506, 2004.

[15] C. Artho, K. Havelund, and A. Biere. High-level Data Races. *Software Testing, Verification and Reliability*, 13(4):207–227, 2003.

[16] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Practical Fault Localization for Dynamic Web Applications. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 265–274. ACM/IEEE, 2010.

[17] P. Arumuga Nainar and B. Liblit. Adaptive Bug Isolation. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 255–264. ACM/IEEE, 2010.

[18] G. Baah, A. Podgurski, and M. Harrold. Causal Inference for Statistical Fault Localization. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA, pages 73–84. ACM, 2010.

[19] R. Baecker, C. DiGiano, and A. Marcus. Software Visualization for Debugging. *Communications of the ACM*, 40(4):44–54, 1997.

[20] T. Ball. The Concept of Dynamic Analysis. In *Proceedings of the European Software Engineering Conference held jointly with the International Symposium on Foundations of Software Engineering*, ESEC/FSE, pages 216–234. Springer-Verlag, 1999.

[21] T. Ball and S. Eick. Software Visualization in the Large. *Computer*, 29(4):33–43, 1996.

[22] T. Ball and J. Larus. Efficient Path Profiling. In *Proceedings of the International Symposium on Microarchitecture*, MICRO, pages 46–57. IEEE Computer Society, 1996.

[23] M. Ballou. Improving Software Quality to Drive Business Agility. Technical report, International Data Corporation (IDC), 2008.

[24] M. Balzer, O. Deussen, and C. Lewerentz. Voronoi Treemaps for the Visualization of Software Metrics. In *Proceedings of the Symposium on Software Visualization*, SoftVis, pages 165–172. ACM, 2005.

[25] O. Barzilay, O. Hazzan, and A. Yehudai. Characterizing Example Embedding as a Software Activity. In *Proceedings of the Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, SUITE, pages 5–8. IEEE Computer Society, 2009.

[26] O. Baysal, M. Godfrey, and R. Cohen. A Bug You Like: A Framework for Automated Assignment of Bugs. In *Proceedings of the International Conference on Program Comprehension*, ICPC, pages 297–298. IEEE Computer Society, 2009.

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

[28] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change.* Addison-Wesley Professional, 2nd edition, 2004.

[29] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold Co., 2nd edition, 1990.

[30] C. Bennett, D. Myers, M.A. Storey, D.M. German, D. Ouellet, M. Salois, and P. Charland. A Survey and Evaluation of Tool Features for Understanding Reverse-engineered Sequence Diagrams. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(4):291–315, 2008.

[31] A. Bergel, F. Bañados, R. Robbes, and D. Röthlisberger. Spy: A Flexible Code Profiling Framework. *Computer Languages, Systems & Structures*, 38(1):16–28, 2012.

[32] A. Bertolino. Software Testing Research: Achievements, Challenges, Dreams. In *Proceedings of the Conference on Future of Software Engineering*, FOSE, pages 85–103. IEEE Computer Society, 2007.

[33] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What Makes a Good Bug Report? In *Proceedings of the International Symposium on Foundations of Software Engineering*, FSE, pages 308–318. ACM, 2008.

[34] S. Bhansali, W. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for Instruction-level Tracing and Analysis of Program Executions. In *Proceedings of the International Conference on Virtual Execution Environments*, VEE, pages 154–163. ACM, 2006.

[35] J. Bohnet. *Visualization of Execution Traces and its Application to Software Maintenance*. PhD thesis, University of Potsdam, 2010.

[36] J. Bohnet and J. Döllner. Visual Exploration of Function Call Graphs for Feature Location in Complex Software Systems. In *Proceedings of the Symposium on Software Visualization*, SoftVis, pages 95–104. ACM, 2006.

[37] B. Boothe. Efficient Algorithms for Bidirectional Debugging. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI, pages 299–310. ACM, 2000.

[38] G. Bracha and D. Griswold. Strongtalk: Type Checking Smalltalk in a Production Environment. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA, pages 215–230. ACM, 1993.

[39] A. Bragdon, S. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. LaViola. Code Bubbles: Rethinking the User Interface Paradigm of Integrated Development Environments. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 455–464. ACM/IEEE, 2010.

[40] J. Brant, B. Foote, R. Johnson, and D. Roberts. Wrappers to the Rescue. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP, pages 396–417. Springer, 1998.

[41] G. Canfora and L. Cerulo. Supporting Change Request Assignment in Open Source Development. In *Proceedings of the Symposium on Applied Computing*, SAC, pages 1767–1772. ACM, 2006.

[42] M. Carrillo-Castellon, J. Garcia-Molina, E. Pimentel, and I. Repiso. Design by Contract in Smalltalk. *Journal of Object Oriented Programming*, 8(7):23–38, 1996.

[43] D. Chapman. Backstep: Incorporating Reverse-Execution into the Eclipse Java Debugger. Technical report, University of Auckland, 2008.

[44] S. Charters, N. Thomas, and M. Munro. The End of the Line for Software Visualisation? In *In Proceedings of the International Workshop on Visualizing Software for Analysis and Understanding*, VISSOFT, pages 110–112. IEEE Computer Society, 2003.

[45] D. Chelimsky, D. Astels, Z. Dennis, A. Hellesoy, B. Helmkamp, and D. North. *The RSpec Book: Behaviour-Driven Development with RSpec, Cucumber, and Friends*. The Pragmatic Programmer, 2nd edition, 2010.

[46] P. Chevalley and P. Thévenod-Fosse. A Mutation Analysis Tool for Java Programs. *International Journal on Software Tools for Technology Transfer*, 5(1):90–103, 2003.

[47] T. Chilimbi, B. Liblit, K. Mehra, A. Nori, and K. Vaswani. HOLMES: Effective

Statistical Debugging via Efficient Path Profiling. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 34–44. ACM/IEEE, 2009.

[48] J. Choi and H. Srinivasan. Deterministic Replay of Java Multithreaded Applications. In *Proceedings of the Symposium on Parallel and Distributed Tools*, SPDT, pages 48–59. ACM, 1998.

[49] H. Cleve and A. Zeller. Locating Causes of Program Failures. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 342–351. ACM/IEEE, 2005.

[50] J. Cook. Reverse Execution of Java Bytecode. *The Computer Journal*, 45(6):608–619, 2002.

[51] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. van Wijk, and A. van Deursen. Understanding Execution Traces Using Massive Sequence and Circular Bundle Views. In *Proceedings of the International Conference on Program Comprehension*, ICPC, pages 49–58. IEEE Computer Society, 2007.

[52] B. Cornelissen, A. van Deursen, L. Moonen, and A. Zaidman. Visualizing Testsuites to Aid in Software Understanding. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, CSMR, pages 213–222. IEEE Computer Society, 2007.

[53] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A Systematic Survey of Program Comprehension through Dynamic Analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

[54] D. Čubranić and G. Murphy. Automatic Bug Triage Using Text Categorization. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering*, SEKE, pages 92–97. KSI Press, 2004.

[55] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight Defect Localization for Java. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP, pages 528–550. Springer, 2005.

[56] W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution Patterns in Object-Oriented Visualization. In *Proceedings of the Conference on Object-Oriented Technologies and Systems*, COOTS, pages 16–16. USENIX Association, 1998.

[57] R. DeLine, A. Bragdon, K. Rowan, J. Jacobsen, and S. Reiss. Debugger Canvas: Industrial Experience with the Code Bubbles Paradigm. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 1064–1073. ACM/IEEE, 2012.

[58] M. Denker, O. Greevy, and M. Lanza. Higher Abstractions for Dynamic Analysis. In *Proceedings of the International Workshop on Program Comprehension through*

Dynamic Analysis, PCODA, pages 32–38. Technical report 2006-11 Universiteit Antwerpen, 2006.

[59] S. Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software.* Springer, 1st edition, 2007.

[60] M. Dowson. The Ariane 5 Software Failure. *ACM SIGSOFT Software Engineering Notes*, 22(2):84, 1997.

[61] M. Ducassé. Coca: An Automated Debugger for C. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 504–513. ACM/IEEE, 1999.

[62] S. Ducasse, M. Lanza, and R. Bertuli. High-Level Polymetric Views of Condensed Run-time Information. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, CSMR, pages 309–318. IEEE Computer Society, 2004.

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

[64] S. Ducasse, L. Renggli, D. Shaffer, R. Zaccone, and M. Davies. *Dynamic Web Development with Seaside.* Self-publishing company, 1st edition, 2010.

[65] A. Dunsmore, M. Roper, and M. Wood. Object-oriented Inspection in the Face of Delocalisation. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 467–476. ACM/IEEE, 2000.

[66] J. Edwards. Example Centric Programming. *ACM SIGPLAN Notices*, 39(12):84–91, 2004.

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

[68] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[69] M. Ernst, A. Czeisler, W. Griswold, and D. Notkin. Quickly Detecting Relevant Program Invariants. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 449–458. ACM/IEEE, 2000.

[70] M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao. The Daikon System for Dynamic Detection of Likely Invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.

[71] S. Feldman and C. Brown. IGOR: A System for Program Debugging via Reversible

Execution. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, PADD, pages 112–123. ACM, 1988.

[72] T. Felgentreff. Comparison, Replay, and Refinement of Communication Traces for Debugging Distributed Failures. Master's thesis, Hasso-Plattner-Institut, 2012.

[73] T. Felgentreff, M. Perscheid, and R. Hirschfeld. Constraining Timing-dependent Communication for Debugging Non-deterministic Failures. In *Proceedings of the Workshop on Academic Software Development Tools and Techniques*, WASDeTT, page accepted. online, 2013.

[74] J. Fierz. Compass: Flow-Centric Back-In-Time Debugging. Master's thesis, University of Bern, 2009.

[75] R. Floyd. Assigning Meanings to Programs. *Mathematical Aspects of Computer Science*, 19(1):19–32, 1967.

[76] T. Fritz, G. Murphy, and E. Hill. Does a Programmer's Activity Indicate Knowledge of Code? In *Proceedings of the European Software Engineering Conference held jointly with the International Symposium on Foundations of Software Engineering*, ESEC/FSE, pages 341–350. ACM, 2007.

[77] T. Fritz, J. Ou, G. Murphy, and E. Murphy-Hill. A Degree-of-Knowledge Model to Capture Source Code Familiarity. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 385–394. ACM/IEEE, 2010.

[78] P. Fritzson, N. Shahmehri, M. Kamkar, and T. Gyimothy. Generalized Algorithmic Debugging and Testing. *ACM Letters on Programming Languages and Systems*, 1(4):303–322, 1992.

[79] Z. Fry and W. Weimer. A Human Study of Fault Localization Accuracy. In *Proceedings of the International Conference on Software Maintenance*, ICSM, pages 1–10. IEEE Computer Society, 2010.

[80] M. Furr, J. An, and J. Foster. Profile-guided Static Typing for Dynamic Scripting Languages. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA, pages 283–300. ACM, 2009.

[81] M. Furr, J. An, J. Foster, and M. Hicks. Static Type Inference for Ruby. In *Proceedings of the Symposium on Applied Computing*, SAC, pages 1859–1866. ACM, 2009.

[82] M. Gaelli. *Modeling Examples to Test and Understand Software.* PhD thesis, University of Bern, 2006.

[83] M. Gaelli, M. Lanza, and O. Nierstrasz. Towards a Taxonomy of Unit Tests. In *Proceedings of the International European Smalltalk Conference*, ESUG, pages 1–10.

Technical report University of Bern, 2005.

[84] M. Gaelli, R. Wampfler, and O. Nierstrasz. Composing Tests from Examples. *Journal of Object Technology*, 6(9):71–86, 2007.

[85] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1st edition, 1995.

[86] T. Gîrba, A. Kuhn, M. Seeberger, and S. Ducasse. How Developers Drive Software Evolution. In *Proceedings of International Workshop on Principles of Software Evolution*, IWPSE, pages 113–122. IEEE Computer Society, 2005.

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

[88] J. Gould. Some Psychological Evidence on How People Debug Computer Programs. *International Journal of Man-Machine Studies*, 7(2):151–182, 1975.

[89] M. Greiler, A. van Deursen, and M. Storey. Test Confessions: A Study of Testing Practices for Plug-in Systems. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 244–254. ACM/IEEE, 2012.

[90] T. Grötker, U. Holtmann, H. Keding, and M. Wloka. *The Developer's Guide to Debugging.* Self-publishing company, 2nd edition, 2012.

[91] T. Gschwind and J. Oberleitner. Improving Dynamic Data Analysis with Aspect-Oriented Programming. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, CSMR, pages 259–268. IEEE Computer Society, 2003.

[92] P. Guerreiro. Simple Support for Design by Contract in C++. In *Proceedings of the International Conference on Technology of Object-Oriented Languages and Systems*, TOOLS, pages 24–34. IEEE, 2001.

[93] L. Gugerty and G. Olson. Comprehension Differences in Debugging by Skilled and Novice Programmers. In *Proceedings of the Workshop on Empirical Studies of Programmers*, ESP, pages 13–27. Ablex, 1986.

[94] P. Guo, T. Zimmermann, N. Nagappan, and B. Murphy. "Not My Bug!" and Other Reasons for Software Bug Report Reassignments. In *Proceedings of the Conference on Computer Supported Cooperative Work*, CSCW, pages 395–404. ACM, 2011.

[95] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating Faulty Code Using Failure-Inducing Chops. In *Proceedings of the International Conference on Automated Software Engineering*, ASE, pages 263–272. ACM/IEEE Computer Society, 2005.

[96] R. Hähnle, M. Baum, R. Bubel, and M. Rothe. A Visual Interactive Debugger

based on Symbolic Execution. In *Proceedings of the International Conference on Automated Software Engineering*, ASE, pages 143–146. ACM/IEEE Computer Society, 2010.

[97] B. Hailpern and P. Santhanam. Software Debugging, Testing, and Verification. *IBM Systems Journal*, 41(1):4–12, 2002.

[98] P. Hamill. *Unit Test Frameworks.* O'Reilly, 1st edition, 2004.

[99] A. Hamou-Lhadj and T. Lethbridge. A Survey of Trace Exploration Tools and Techniques. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON, pages 42–55. IBM Press, 2004.

[100] S. Hangal and M. Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 291–301. ACM/IEEE, 2002.

[101] M. Harrold, G. Rothermel, R. Wu, and L. Yi. An Empirical Investigation of Program Spectra. In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering*, PASTE, pages 83–90. ACM, 1998.

[102] M. Haupt, R. Hirschfeld, T. Pape, G. Gabrysiak, S. Marr, A. Bergmann, A. Heise, M. Kleine, and R. Krahn. The SOM Family: Virtual Machines for Teaching and Research. In *Proceedings of the Conference on Innovation and Technology in Computer Science Education*, ITiCSE, pages 18–22. ACM, 2010.

[103] 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.

[104] C. Hermanns. *Entwicklung und Implementierung eines hybriden Debuggers für Java.* PhD thesis, University of Münster, 2010.

[105] R. Hirschfeld, M. Perscheid, and M. Haupt. Explicit Use-case Representation in Object-oriented Programming Languages. In *Proceedings of the Dynamic Languages Symposium*, DLS, pages 51–60. ACM, 2011.

[106] 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.

[107] A. Ho and S. Hand. On the Design of a Pervasive Debugger. In *Proceedings of the International Symposium on Automated Analysis-driven Debugging*, AADEBUG, pages 117–122. ACM, 2005.

[108] C. Hofer, M. Denker, and S. Ducasse. Design and Implementation of a Backward-

in-Time Debugger. In *Proceedings of NODE*, volume 6, pages 17–32. Springer, 2006.

[109] U. Hölzle and D. Ungar. Optimizing Dynamically-dispatched Calls with Run-time Type Feedback. *ACM SIGPLAN Notices*, 29(6):326–363, 1994.

[110] J. Horgan, S. London, and M. Lyu. Achieving Software Quality with Testing Coverage Measures. *Computer*, 27(9):60–69, 1994.

[111] J. Huffman Hayes, A. Dekhtyar, and D. Janzen. Towards Traceable Test-Driven Development. In *Proceedings of the Workshop on Traceability in Emerging Forms of Software Engineering*, TEFSE, pages 26–30. IEEE Computer Society, 2009.

[112] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA, pages 318–326. ACM, 1997.

[113] T. Janssen, R. Abreu, and A. van Gemund. Zoltar: A Toolset for Automatic Fault Localization. In *Proceedings of the International Conference on Automated Software Engineering*, ASE, pages 662–664. ACM/IEEE Computer Society, 2009.

[114] D. Jeffrey, N. Gupta, and R. Gupta. Fault Localization Using Value Replacement. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA, pages 167–178. ACM, 2008.

[115] G. Jeong, S. Kim, and T. Zimmermann. Improving Bug Triage with Bug Tossing Graphs. In *Proceedings of the European Software Engineering Conference held jointly with the International Symposium on Foundations of Software Engineering*, ESEC/FSE, pages 111–120. ACM, 2009.

[116] D. Jerding, J. Stasko, and T. Ball. Visualizing Message Patterns in Object-Oriented Program Executions. Technical report, Georgia Institute of Technology, 1996.

[117] B. Jiang, Z. Zhang, T. Tse, and T. Chen. How Well Do Test Case Prioritization Techniques Support Statistical Fault Localization. In *Proceedings of the International Computer Software and Applications Conference*, COMPSAC, pages 99–106. IEEE Computer Society, 2009.

[118] L. Jiang and Z. Su. Context-Aware Statistical Debugging: From Bug Predictors to Faulty Control Flow Paths. In *Proceedings of the International Conference on Automated Software Engineering*, ASE, pages 184–193. ACM/IEEE Computer Society, 2007.

[119] B. Johnson and B. Shneiderman. Tree Maps: A Space-filling Approach to the Visualization of Hierarchical Information Structures. In *Proceedings of the Conference on Visualization*, VIS, pages 284–291. IEEE, 1991.

[120] J. Jones, J. Bowring, and M. Harrold. Debugging in Parallel. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA, pages 16–26. ACM, 2007.

[121] J. Jones and M. Harrold. Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. In *Proceedings of the International Conference on Automated Software Engineering*, ASE, pages 273–282. ACM/IEEE Computer Society, 2005.

[122] J. Jones, M. Harrold, and J. Stasko. Visualization of Test Information to Assist Fault Localization. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 467–477. ACM/IEEE, 2002.

[123] H. Kagdi, M. Hammad, and J. Maletic. Who Can Help Me with this Source Code Change? In *Proceedings of the International Conference on Software Maintenance*, ICSM, pages 157–166. IEEE, 2008.

[124] H. Kagdi and D. Poshyvanyk. Who Can Help Me with this Change Request? In *Proceedings of the International Conference on Program Comprehension*, ICPC, pages 273–277. IEEE Computer Society, 2009.

[125] I. Katz and J. Anderson. Debugging: An Analysis of Bug-location Strategies. *Human-Computer Interaction*, 3(4):351–399, 1987.

[126] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP, pages 220–242. Springer, 1997.

[127] S. Kim, J. Clark, and J. McDermid. Class Mutation: Mutation Testing for Object-Oriented Programs. In *Proceedings of the Net. ObjectDays*, pages 9–12. Springer, 2000.

[128] A. Ko and B. Myers. Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 301–310. ACM/IEEE, 2008.

[129] T. Koju, S. Takada, and N. Doi. An Efficient and Generic Reversible Debugger Using the Virtual Machine Based Approach. In *Proceedings of the International Conference on Virtual Execution Environments*, VEE, pages 79–88. ACM, 2005.

[130] R. Kramer. iContract - The Java Design by Contract Tool. In *Proceedings of the International Conference on Technology of Object-Oriented Languages and Systems*, TOOLS, pages 295–307. IEEE, 1998.

[131] A. Kuhn, B. van Rompaey, L. Hänsenberger, O. Nierstrasz, S. Demeyer, M. Gaelli, and K. van Leemput. JExample: Exploiting Dependencies Between Tests to Improve Defect Localization. In *Proceedings of the International Conference on*

Extreme Programming and Agile Processes in Software Engineering, XP, pages 73–82. Springer, 2008.

[132] M. Lanza, R. Marinescu, and S. Ducasse. *Object-oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-oriented Systems.* Springer, 1st edition, 2006.

[133] R. Lencevicius, U. Hölzle, and A. Singh. Query-based Debugging of Object-Oriented Programs. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA, pages 304–317. ACM, 1997.

[134] R. Lencevicius, U. Hölzle, and A. Singh. Dynamic Query-Based Debugging. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP, pages 135–160. Springer-Verlag, 1999.

[135] B. Lewis. Debugging Backwards in Time. In *Proceedings of the International Workshop on Automated Debugging*, AADEBUG, pages 225–235. Arxiv, 2003.

[136] 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.

[137] 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.

[138] H. Lieberman. The Debugging Scandal and What to Do About It. *Communications of the ACM*, 40(4):26–29, 1997.

[139] H. Lieberman and C. Fry. ZStep 95: A Reversible, Animated Source Code Stepper. *Software Visualization: Programming as a Multimedia Experience*, pages 277–292, 1997.

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

[141] A. Lienhard. *Dynamic Object Flow Analysis.* PhD thesis, University of Bern, 2008.

[142] A. Lienhard, T. Gîrba, and O. Nierstrasz. Practical Object-Oriented Back-in-Time Debugging. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP, pages 592–615. Springer, 2008.

[143] C. Liu, X. Yan, L. Fei, J. Han, and S. Midkiff. SOBER: Statistical Model-based Bug Localization. In *Proceedings of the European Software Engineering Conference held jointly with the International Symposium on Foundations of Software Engineering*, ESEC/FSE, pages 286–295. ACM, 2005.

[144] W. Löwe, A. Ludwig, and A. Schwind. Understanding Software–Static and Dynamic Aspects. In *Proceedings of the International Conference on Advanced Science and Technology*, ICAST, pages 52–57, 2001.

[145] H. Lü and J. Fogarty. Cascaded Treemaps: Examining the Visibility and Stability of Structure in Treemaps. In *Proceedings of Graphics Interface*, GI, pages 259–266. Canadian Information Processing Society, 2008.

[146] D. Matter, A. Kuhn, and O. Nierstrasz. Assigning Bug Reports Using a Vocabulary-based Expertise Model of Developers. In *Proceedings of the International Working Conference on Mining Software Repositories*, MSR, pages 131–140. IEEE Computer Society, 2009.

[147] T. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[148] D. McDonald and M. Ackerman. Expertise Recommender: A Flexible Recommendation System and Architecture. In *Proceedings of the Conference on Computer Supported Cooperative Work*, CSCW, pages 231–240. ACM, 2000.

[149] G. Meszaros. *XUnit Test Patterns: Refactoring Test Code.* Prentice Hall, 1st edition, 2006.

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

[151] B. Meyer. Applying “Design by Contract”. *Computer*, 25(10):40–51, 1992.

[152] S. Minto and G. Murphy. Recommending Emergent Teams. In *Proceedings of the International Workshop on Mining Software Repositories*, MSR, pages 5–12. IEEE Computer Society, 2007.

[153] J. Misurda, J. Clause, J. Reed, B. Childers, and M. Soffa. Demand-Driven Structural Testing with Dynamic Instrumentation. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 156–165. ACM/IEEE, 2005.

[154] A. Mockus and J. Herbsleb. Expertise Browser: A Quantitative Approach to Identifying Expertise. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 503–512. ACM/IEEE, 2002.

[155] G. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing.* John Wiley & Sons, 3rd edition, 2011.

[156] K. Nadiminti, M. De Assunção, and R. Buyya. Distributed systems and recent innovations: Challenges and benefits. *InfoNet Magazine*, 16:1–5, 2006.

[157] L. Naish. A Declarative Debugging Scheme. *Journal of Functional and Logic*

Programming, 1997(3):1–16, 1997.

[158] K. Nørmark. Systematic Unit Testing in a Read-eval-print Loop. *Journal of Universal Computer Science*, 16(2):296–314, 2010.

[159] The Institute of Electrical and Electronics Engineers. *IEEE Standard Glossary of Software Engineering Terminology.* IEEE, 1st edition, 1990.

[160] A. Orso, J. Jones, and M. Harrold. Visualization of Program-Execution Data for Deployed Software. In *Proceedings of the Symposium on Software Visualization*, SoftVis, pages 67–76. ACM, 2003.

[161] M. Pacione, M. Roper, and M. Wood. A Comparative Evaluation of Dynamic Visualisation Tools. In *Proceedings of the Working Conference on Reverse Engineering*, WCRE, pages 80–89. IEEE Computer Society, 2003.

[162] 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.

[163] S. Park, R. Vuduc, and M. Harrold. Falcon: Fault Localization in Concurrent Programs. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 245–254. ACM/IEEE, 2010.

[164] J. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W. Wong, Y. Zibin, M. Ernst, and M. Rinard. Automatically Patching Errors in Deployed Software. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP, pages 87–102. ACM, 2009.

[165] 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, pages 60–67. IEEE, 2012.

[166] 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.

[167] 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.

[168] 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.

[169] 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.

[170] R. Plösch. Design By Contract for Python. In *Proceedings of the Asia Pacific and International Software Engineering Conference*, APSEC, pages 213–219. IEEE, 1997.

[171] F. Pluquet, A. Marot, and R. Wuyts. Fast Type Reconstruction for Dynamically Typed Programming Languages. In *Proceedings of the Dynamic Languages Symposium*, DLS, pages 69–78. ACM, 2009.

[172] N. Polikarpova, I. Ciupa, and B. Meyer. A Comparative Study of Programmer-written and Automatically Inferred Contracts. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA, pages 93–104. ACM, 2009.

[173] A. Potanin, J. Noble, and R. Biddle. Snapshot Query-Based Debugging. In *Proceedings of the Australian Software Engineering Conference*, ASWEC, pages 251–259. IEEE Computer Society, 2004.

[174] G. Pothier and É. Tanter. Summarized Trace Indexing and Querying for Scalable Back-in-Time Debugging. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP, pages 558–582. Springer, 2011.

[175] G. Pothier, É. Tanter, and J. Piquer. Scalable Omniscient Debugging. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA, pages 535–552. ACM, 2007.

[176] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. Reiss. Automated Fault Localization Using Potential Invariants. In *In Proceedings of the International Workshop on Automated and Algorithmic Debugging*, AADEBUG, pages 273–276. arXiv Preprint, 2003.

[177] 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.

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

[179] S. Reiss and M. Renieris. Encoding Program Executions. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 221–230. ACM/IEEE, 2001.

[180] M. Renieres and S. Reiss. Fault Localization with Nearest Neighbor Queries. In *Proceedings of the International Conference on Automated Software Engineering*, ASE, pages 30–39. ACM/IEEE Computer Society, 2003.

[181] J. Ressia, A. Bergel, and O. Nierstrasz. Object-Centric Debugging. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 485–495. ACM/IEEE, 2012.

[182] D. Richardson and M. Thompson. An Analysis of Test Data Selection Criteria Using the RELAY Model of Fault Detection. *IEEE Transactions on Software Engineering*, 19(6):533–553, 1993.

[183] D. Röthlisberger, O. Greevy, and O. Nierstrasz. Exploiting Runtime Information in the IDE. In *Proceedings of the International Conference on Program Comprehension*, ICPC, pages 63–72. IEEE Computer Society, 2008.

[184] D. Röthlisberger, M. Härry, A. Villazón, D. Ansaloni, W. Binder, O. Nierstrasz, and P. Moret. Augmenting Static Source Views in IDEs with Dynamic Metrics. In *Proceedings of the International Conference on Software Maintenance*, ICSM, pages 253–262. IEEE Computer Society, 2009.

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

[186] D. Saff and M. Ernst. Continuous Testing in Eclipse. *Electronic Notes in Theoretical Computer Science*, 107:103–117, 2004.

[187] Y. Saito. Jockey: A User-space Library for Record-replay Debugging. In *Proceedings of the International Symposium on Automated Analysis-driven Debugging*, AADEBUG, pages 69–76. ACM, 2005.

[188] K. Sakurai, H. Masuhara, and S. Komiya. Traceglasses: A Trace-based Debugger for Realizing Efficient Navigation. *IPSJ Special Interest Group on Programming*, 3(3):1–17, 2010.

[189] R. Santelices, J. Jones, Y. Yu, and M. Harrold. Lightweight Fault-Localization Using Multiple Coverage Types. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 56–66. ACM/IEEE, 2009.

[190] B. Schneiderman and C. Plaisant. *Designing the User Interface.* Addison-Wesley, 3rd edition, 1998.

[191] D. Schuler and T. Zimmermann. Mining Usage Expertise from Version Archives. In *Proceedings of the International Working Conference on Mining Software Repositories*, MSR, pages 121–124. ACM, 2008.

[192] F. Servant and J. Jones. WhoseFault: Automatic Developer-to-Fault Assignment through Fault Localization. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 36–46. ACM/IEEE, 2012.

[193] E. Shapiro. *Algorithmic Program Debugging.* PhD thesis, Yale University, 1982.

[194] B. Shneiderman and M. Wattenberg. Ordered Treemap Layouts. In *Proceedings of the Symposium on Information Visualization*, INFOVIS, pages 73–78. IEEE, 2001.

[195] S. Spoon and O. Shivers. Demand-Driven Type Inference with Subgoal Pruning: Trading Precision for Scalability. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP, pages 485–493. Springer, 2004.

[196] B. Steinert, M. Haupt, R. Krahn, and R. Hirschfeld. Continuous Selective Testing. In *Proceedings of the Conference on Agile Processes in Software Engineering and Extreme Programming*, XP, pages 132–146. Springer, 2010.

[197] 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.

[198] N. Suzuki. Inferring Types in Smalltalk. In *Proceedings of the Symposium on Principles of Programming Languages*, POPL, pages 187–199. ACM, 1981.

[199] L. Thamsen, A. Gulenko, M. Perscheid, R. Krahn, R. Hirschfeld, and D. Thomas. Orca: A Single-language Web Framework for Collaborative Development. In *Proceedings of the Conference on Creating, Connecting and Collaborating through Computing*, C5, pages 45–52. IEEE, 2012.

[200] A. Tolmach and A. Appel. A Debugger for Standard ML. *Journal of Functional Programming*, 5(2):155–200, 1995.

[201] D. Ungar, H. Lieberman, and C. Fry. Debugging and the Experience of Immediacy. *Communications of the ACM*, 40(4):38–43, 1997.

[202] A. van Deursen. Program Comprehension Risks and Opportunities in Extreme Programming. In *Proceedings of the Working Conference on Reverse Engineering*, WCRE, pages 176–185. IEEE Computer Society, 2001.

[203] A. van Deursen, L. Moonen, A. Bergh, and G. Kok. Refactoring Test Code. Technical report, Centrum Wiskunde & Informatica, 2001.

[204] J. van Geet, A. Zaidman, O. Greevy, and A. Hamou-Lhadj. A Lightweight Approach to Determining the Adequacy of Tests as Documentation. In *Proceedings of the International Workshop on Program Comprehension through Dynamic Analysis*, PCODA, pages 21–26. Technical report 2006-11 Universiteit Antwerpen, 2006.

[205] C. van Rijsbergen. *Information Retrieval.* Butterworth, 1st edition, 1979.

[206] B. van Rompaey and S. Demeyer. Establishing Traceability Links between Unit Test Cases and Units under Test. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, CSMR, pages 209–218. IEEE Computer

Society, 2009.

[207] J. van Wijk and H. van de Wetering. Cushion Treemaps: Visualization of Hierarchical Information. In *Proceedings of the Symposium on Information Visualization*, INFOVIS, pages 73–78. IEEE, 1999.

[208] S. Vegas, N. Juristo, and V. Basili. Maturing Software Engineering Knowledge through Classifications: A Case Study on Unit Testing Techniques. *IEEE Transactions on Software Engineering*, 35(4):551–565, 2009.

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

[210] I. Vessey. Toward a Theory of Computer Program Bugs: An Empirical Test. *International Journal of Man-Machine Studies*, 30(1):23–46, 1989.

[211] E. Wilson. *An Introduction to Scientific Research*. McGraw-Hill, 1st edition, 1952.

[212] T. Xie and D. Notkin. Automatic Extraction of Object-oriented Observer Abstractions from Unit-Test Executions. In *In Proceedings of the International Conference on Formal Engineering Methods*, ICFEM, pages 290–305. Springer, 2004.

[213] Q. Yang, J. Li, and D. Weiss. A Survey of Coverage-Based Testing Tools. *The Computer Journal*, 52(5):589–597, 2009.

[214] C. Yilmaz, A. Paradkar, and C. Williams. Time Will Tell: Fault Localization Using Time Spectra. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 81–90. ACM/IEEE, 2008.

[215] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How Do Fixes Become Bugs? In *Proceedings of the European Software Engineering Conference held jointly with the International Symposium on Foundations of Software Engineering*, ESEC/FSE, pages 26–36. ACM, 2011.

[216] Y. Yu, J. Jones, and M. Harrold. An Empirical Study of the Effects of Test-suite Reduction on Fault Localization. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 201–210. ACM/IEEE, 2008.

[217] A. Zeller. Isolating Cause-effect Chains from Computer Programs. In *Proceedings of the International Symposium on Foundations of Software Engineering*, FSE, pages 1–10. ACM, 2002.

[218] A. Zeller. Program Analysis: A Hierarchy. In *Proceedings of the Workshop on Dynamic Analysis*, WODA, pages 6–9. http://www.cs.nmsu.edu/ jcook/woda2003/ (last access: 18 August 2012), 2003.

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

2nd edition, 2009.

[220] A. Zeller and R. Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[221] A. Zeller and D. Lütkehaus. DDD—A Free Graphical Front-End for UNIX Debuggers. *ACM SIGPLAN Notices*, 31(1):22–27, 1996.

[222] X. Zhang, N. Gupta, and R. Gupta. Locating Faults through Automated Predicate Switching. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 272–281. ACM/IEEE, 2006.