# Understanding Service Implementations Through Behavioral Examples

Michael Perscheid
michael.perscheid@hpi.uni-potsdam.de

The understanding of service implementations, with a special focus on internal details that constitute functionality, is an important aspect during the development of services. Actual run-time data supports the comprehension of service implementations like examples support the explanation of abstract concepts and principles. However, the required run-time analysis is often associated with an inconvenient overhead that renders current tools impractical for frequent use.

We propose a practical, lightweight, and incremental approach to dynamic analysis based on entry points that describe reproducible system behavior. By observing and enriching such concrete examples of behavioral paths, we investigate new perspectives on service implementations that improve comprehension of execution semantics. We are providing perspectives for exploring one execution path in detail, for comparing multiple paths with each other, and for understanding the internal behavior from the users point of view.

## 1 Introduction

Developers of object-oriented software systems, including service implementations, spend a significant amount of time on program comprehension [4]. They require an in-depth understanding of the code base that they work on; ranging from the intended use of an interface to the collaboration of objects. Gaining an understanding of a program by reading source code alone is difficult as it is inherently abstract.

Run-time information supports developers in coping with this complexity. Collected run-time data reports on the effects of source code and thus helps understanding it. At run-time, the abstract gets concrete; variables refer to concrete objects and messages get bound to concrete methods. For example, a program's run-time helps to answer: "How is a particular method called?" or "How does the value of a variable change?"

Unfortunately, the overhead imposed by current tools renders them impractical for frequent use. This is mainly due to two issues; setting up an analysis tool usually requires a significant configuration effort and performing the required in-depth dynamic analysis is time-consuming. Both issues inhibit immediacy and thus discourage developers from using these tools frequently.

We propose a new approach to dynamic analysis that enables a feeling of immediacy that current tools are missing. Based on entry points that describe reproducible behavior, we split the costs of dynamic analysis over multiple runs—an initial shal-

low analysis followed by detached in-depth on-demand refinements. For our implementation, we leverage test cases as such entry points, as they commonly satisfy the necessary requirements. Triggered by their execution, we observe and enrich behavioral examples to improve the comprehension of execution semantics. We develop the Path tool suite, which allows developers to explore a specific execution path in detail (PathFinder), to compare multiple paths with each other (PathMap), and to understand the internals from the user's point of view (PathTrace).

The remainder of this paper is organized as follows: Section 2 presents our approach to dynamic analysis that collects data exactly when needed. Section 3 describes the Path tool suite. Section 4 demonstrates how program comprehension is supported by our tools. Section 5 concludes and presents next steps.

# 2 Dynamic Service Analysis

Dynamic service analysis is a practical, lightweight, and incremental approach to dynamic analysis for understanding system and service implementations. It is build on reproducible entry points such as test cases that act as concrete examples of service behavior. These behavioral examples concrete the abstract entities of source code with meaningful information and so support program comprehension.

By executing entry points multiple times, we can split the dynamic analysis costs over multiple runs. Starting with an initial shallow analysis for navigating behavioral paths, developers get only a subset of all possible run-time data. When they require more detailed information such as object states, this additional data is collected in detached on-demand refinements by executing entry points multiple times. This approach enables a feeling of immediacy that current dynamic analysis concepts are missing.

For an easy setup of analysis tools, our approach can seamlessly be integrated into current development environments. Developers only have to define their subsystem of interest and a collection of entry points. While the first must be done with the declaration of packages only once, the latter can be automized if test cases or API examples are available. Furthermore, by continuously maintaining a coverage relationship between executed entry points and covered methods [12], it is possible to embed arbitrary methods into meaningful examples at all times.

Figure 1 summarizes our approach. First, reproducible entry points such as test cases are executed to produce behavioral example paths (Section 2.1). Second, dynamic analysis is applied to these examples for exploring a specific path in detail or comparing multiple paths with each other (Section 2.2). Finally, the meaning of entry points can be enhanced with links to requirements for recovering traceability information automatically (Section 2.3).

## 2.1 Entry Point Characteristics

Dynamic service analysis requires the ability to reproduce arbitrary points in a program execution. Therefore, we assume the existence of entry points that specify deterministic program executions. For our implementation, we leverage test cases as such entry

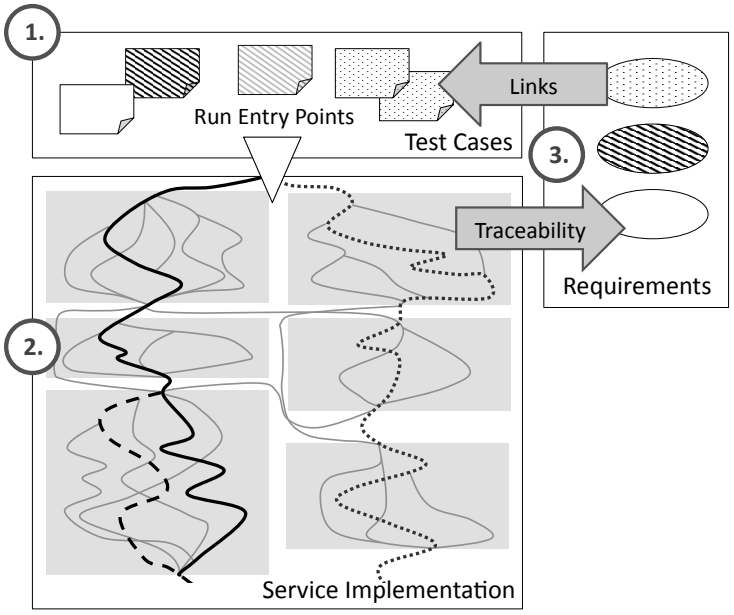


Figure 1: Dynamic service analysis allows for exploring behavioral paths through service implementations. Triggered by the run of entry points (here test cases) (1), we analyze specific or multiple paths (2) to support program comprehension. Furthermore, by linking entry points and requirements (3), we recover traceability data automatically.

points, as they commonly satisfy this requirement [13]—a test case describes what the system is expected to do, its execution reveals how it is realized. Moreover, tests are meaningful behavioral examples of the system and its parts; they can be executed repeatedly, fast, and without any side effect before and after their execution; and they are an integral part of several development processes, especially agile processes.

Leveraging test cases as entry points is not a requirement for dynamic service analysis. Our tools work best if test coverage for the developed application is high, but resorts to manually specified entry points if no covering test is found. This, however, requires more knowledge about the system under observation than relying on test coverage: it is not always trivial to anticipate control flows leading to methods of interest.

## 2.2 Step-wise Run-time Analysis

Traditional approaches to dynamic analysis are time-consuming as they capture comprehensive information about the entire execution up-front. Low costs can be achieved by structuring program analysis according to user needs or, more specifically, dividing the analysis into multiple steps.

Step-wise run-time analysis [8] splits the analysis of a program's run-time over multiple runs: A high-level analysis followed by on-demand refinements. A first *shallow analysis* focuses on the information that is required for presenting an overview of a program run. Further information about method arguments or instance variables are

not recorded. As the developer identifies relevant details, such data is recorded on-demand in additional *refinement analysis* runs. The information required for program comprehension 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 costs by loading information only when the user identifies interest. This provides for quick access to relevant run-time information without collecting needless data.

The concept of step-wise run-time analysis is also adaptable to multiple execution paths. Instead of running only one entry point, several entry points can also be executed and analyzed one after another. Only required run-time information is collected and if necessary summarized with dynamic metrics. In this way, behavioral paths, different states, or coverage data can be compared with each other. Analogous to the dynamic analysis of a specific path, we collect only initial run-time information for the task at hand. When more detailed information is required, it can be refined in detached runs. Thus, dynamic analysis can also be divided into fine-granular and incremental steps for multiple entry points and their execution paths.

## 2.3 Link Entry Points, Get Traceability

Additional links between entry points and their primary objectives support program comprehension as developers can understand the reasons for exemplary system behavior. So far, entry points offer examples into system behavior but their real purpose is hidden in external requirements, the development history, or implementation artifacts. Especially, test cases were implemented with a particular reason in mind such as verifying a specific requirement. With the connection between entry points and requirements (or other development entities), the meaning of subsequently behavioral paths can be enhanced.

Links between entry points and requirements in combination with dynamic service analysis allow for recovering traceability information automatically [6]. Requirements traceability is considered to be important for software understanding as it supports answers for questions such as why a particular source code entity was implemented. Adapting the concepts of feature localization [2], annotated entry points are executed and behavioral paths are related to entry points and requirements in question. Afterwards a requirement is traced to a source code entity if it has been executed at least once in a specific entry point that is linked to this requirement. Depending on the coverage of entry points, large parts of the system can be traced and classified to requirements automatically.

Future work deals with the automatization of the manual linking step. We are working on the integration of acceptance test frameworks with dynamic service analysis. As some acceptance test frameworks store or explicitly represent the relation to tested requirements, this information can be reused for replacing the manual linking process. Consequently, we have a fully automatic requirements traceability approach from requirements via dynamic analysis of tests through to traced source code entities.

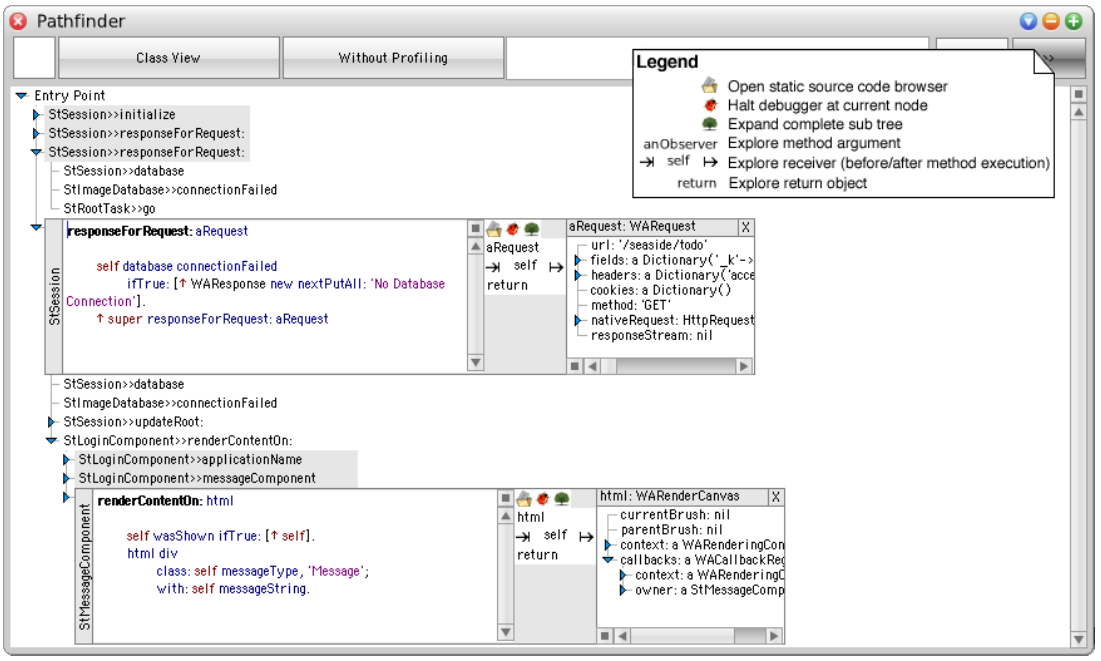


Figure 2: PathFinder is our interactive dynamic analysis tool that allows developers to explore a specific behavioral path through the service (ToDo Web application [9]).

# 3 An Overview of the Path Tool Suite

The *path tool suite* realizes the concepts of dynamic service analysis and supports the understanding of behavior. By means of examples, being triggered by reproducible entry points, our tools observe, analyze, and present behavioral *paths* for developers. At the same time, our tools distribute the costs of dynamic analysis across multiple runs and so enable a feeling of immediacy when program behavior is explored—a property that cannot be provided by fully dynamic analysis approaches.

Our tools are combined with each other in order to refine behavioral questions step by step and from different perspectives. PathFinder (Section 3.1) reveals one specific behavioral path, its state, and object interactions. PathMap (Section 3.2) summarizes run-time information with the help of several metrics and presents it inside the system's architecture. PathTrace (Section 3.3) connects the user's and developer's point of view as it traces requirements down to source code entities. At the end, all tools are supposed to answer behavioral questions for one or multiple execution paths from several points of view—a characteristic that is only minimal supported by current tools [5, 11].

## 3.1 PathFinder: Interactive Dynamic Views

PathFinder [8] is our dynamic analysis tool for interactively introspecting the behavior of one specific execution path. Based on a lightweight call graph representation used for navigation (shallow analysis), developers can state their points of interest and all further information is computed on demand by re-executing the chosen entry point in

background (refinement analysis). We distribute the overhead of dynamic analysis over multiple runs so that there is no need to collect all data at once. Thereby memory consumption and performance impacts are kept low. Thus, we have a practical approach for behavioral views that will be evaluated regarding its improvements for program comprehension in the near future.

Figure 2 presents PathFinder while a service request of our ToDo Web application [9] is processed. The internal behavior is revealed for this specific request and developers can follow what inside the application happens. If they are interested in further details, the request is executed again and more detailed information such as method arguments is collected and presented. Thus, behavioral reachability questions concerning only one execution path [5, 11], for instance, ”what happens before executing the `renderContentOn:` method?” or ”how does the request object change?”, can be answered with the help of PathFinder. Its interactive dynamic views allow for navigating call and object trees in both forward and backward direction so that developers get insights into the execution and state history without any restrictions. Furthermore, almost all run-time information including the initial shallow analysis is provided in less than a second (less than 300 milliseconds for 95% of about 4.400 test cases [8]). These features should encourage developers to use PathFinder at least as often as they use tedious and time-consuming debugging strategies for program comprehension [5].

## 3.2 PathMap: What We Can Learn from Tests

PathMap summarizes multiple behavioral paths and merges static and dynamic views of a system under observation. In contrast to PathFinder, it has a stronger focus on test cases as entry points but the presented concepts are still independent of that fact. PathMap is integrated into a standard test runner and enhances the value of running test cases by analyzing their execution, rendering dynamic metrics, and suggesting meaningful entities, which can be further explored with the aforementioned PathFinder tool. We investigate new system perspectives that are intended to support several software engineering tasks such as guiding developers to potential locations for traced requirements, hot spots, or untested code. Moreover, we compare dynamic paths and reveal anomalies for more suitable fault localization or hints at design disharmonies.

Figure 3 illustrates the PathMap and the test quality analysis of the Seaside Web framework [1]. Seaside’s system structure is rendered within a tree map layout where packages include classes which in turn include method entities. The entire Web framework (more than 3,700 methods) is presented in the space of a 500 pixel square, which can be explored interactively. Furthermore, the static view can be colorized with a static metric such as lines of code, complexity or as in this example the developer of a method. Having analyzed the execution of all test cases, PathMap darken method entities that has been covered by at least on entry point. As a consequence in this example, we are able to answer the questions ”who has written insufficiently tested methods?” or ”which subsystems need more attention during quality assurance?”.

PathMap provides answers to behavioral reachability questions that deal with multiple execution paths [5, 11]. We support the following scenarios:

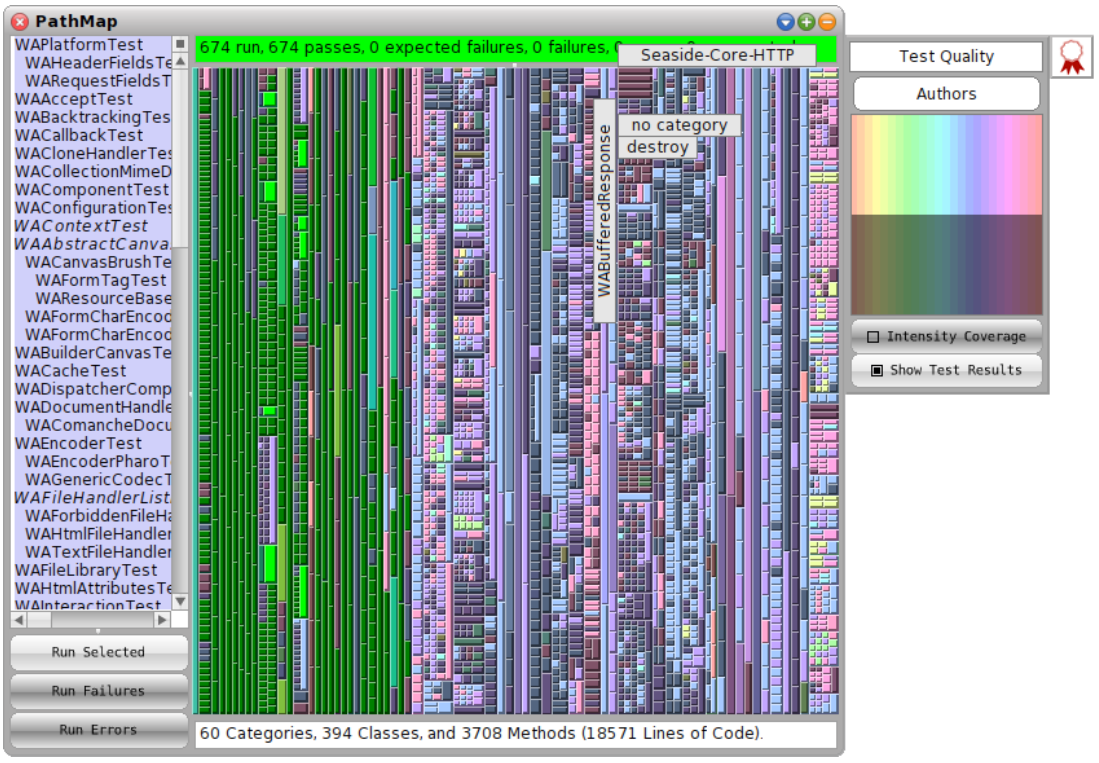


Figure 3: PathMap executes multiple entry points and summarizes the analyzed behavior within the system's architecture (Seaside Web framework [1] - Test coverage in relation to the developer of a method).

**Test Quality** Combining arbitrary static metrics with coverage information allows for assessing software quality in terms of application structure and testing activities.

**Concerns Traceability** Based on the concept of feature localization [2], we trace arbitrary concerns to covered entities and support developers in comprehending the system from different points of view.

**Fault Localization** With the comparison of passed and failed test cases [10], failures can be localized more closely to real causes.

**Profiling** Instead of looking at one execution profile, we summarize multiple executions and identify hot spots for processing time, number of objects, or method calls.

## 3.3 PathTrace: Understanding the User's Point of View

PathTrace is a semi-automated approach for the post-traceability of requirements that allows developers to comprehend the system from the users point of view. Based on the concepts of feature localization, we manually link entry points with requirements, analyze their behavioral paths, and trace requirements to covered source code entities


Figure 4: AcceptIt allows for writing acceptance tests in a business-readable domain specific language so that requirement descriptions and related tests are one and the same. Based on such entities, PathTrace can automatically recover traceability data.

automatically [6]. Other development tools can use the gathered traceability information to improve understanding of user requests and their related implementation details. Thus, answers for typical software maintenance questions such as ”how is this requirement implemented?” or ”which source code entities are related to the user’s failure report?” [11] are better supported.

To complete our approach to a fully automatic traceability technique, we are working on a new acceptance test framework called AcceptIt[1]. AcceptIt provides a way of merging requirements and tests into one executable source code entity. In other words, linking becomes redundant as links are implicitly available. Tests are described in a business-readable domain specific language (BR-DSL). This BR-DSL is mapped to ordinary library methods. In consequence, acceptance tests are executable as usual and readable (maybe also writable) by customers. The difference between an acceptance test and a scenario description becomes blurred.

Figure 4 presents the current state of AcceptIt with its BR-DSL, its language extension, and a part of the source code library (example taken from our ToDo Web application [9]). On the left side, the ”login” scenario as part of a user story is defined and can be understood without any knowledge of the implementation—`given` declares preconditions, `when` describes actions, and `then` asserts a specific event or state. On the right side, the `given` step is mapped to the corresponding library method and its implementation. At a later time, libraries should offer a lot of generic mappings so that they are reusable in several scenarios.

AcceptIt introduces a new kind of entry point. On the one hand PathTrace is extended to be completely automatic on the other hand PathFinder and PathMap analyze entry points with clear objectives. So, developers have not only the answer to what a

[1]The idea is based on Cucumber `http://www.cukes.info`. Last accessed: October 8, 2010

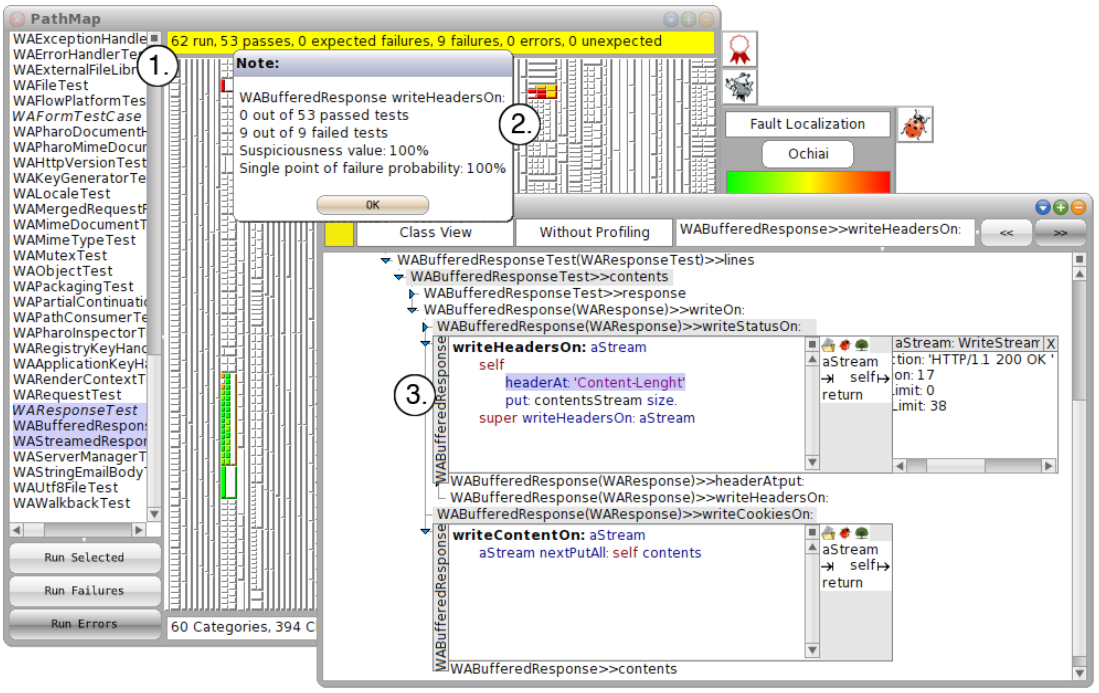


Figure 5: After selecting and running failing tests (1), PathMap compares test run results and colorizes suspicious methods (2). Subsequently, PathFinder allows for exploring a specific path to the cause of a failure (3).

test case describes and how it is realized, they have also its purpose. In the sense of program comprehension, we can enhance the meaning of behavioral paths.

# 4 Case Study: Fault Localization in Seaside

For demonstrating dynamic service analysis and our Path tool suite, we have done a case study about fault localization in the Seaside Web framework [1]. Since fault localization or in other words ”debugging” usually requires much knowledge about system behavior, it is an ideal use case for presenting the benefits of our approach.

We have introduced a defect into Seaside’s Web server—inside the header creation of buffered responses, we inserted a ”Content-Lenght” typo. For that reason, service requests with buffered responses produced invalid results but streamed responses still worked correctly. Seaside’s test suite answered with 9 failed and 53 passed test cases for all response tests. Starting the standard debugger at an failed test led to a violated assertion within the test itself. This means that the execution history was lost and the failure cause was not comprehensible. The assertion was thrown for the whole response object without any pointers to the invalid state. The typo was far away from the observable malfunction. Such a situation is the rule rather than the exception [14].

Figure 5 shows how the defect could be identified with the help of our Path tools. First, PathMap replaced the standard test runner, rendered the whole Seaside sys-

tem within a tree map, executed the response tests, analyzed the behavioral paths of passed and failed test cases with the ”Ochiai” metric [10], and colorized the map with suspiciousness and confidence values. In short a method has a higher suspiciousness (hue towards red value) if it was executed in more failing than passing tests and a method has less confidence (brighter color) if it was not executed in all failing tests. Second, the computed values suggested only three methods in full red (100% suspiciousness and 100% confidence). All other methods were paler and not so hot. Third, for understanding all three methods, we opened PathFinder on a failed test as we expected that all three methods (same class) were related to each other. PathFinder showed that the methods `writeContentsOn:` and `writeHeadersOn:` were called in sequence within `writeOn:` (here, we ignored the third method as it was only a simple accessor). While using PathFinder, we navigated the execution history in both directions and with the help of on-demand refinements we verified assumptions concerning corrupted object states. For instance, the method argument `aStream` already included a valid response code. On closer examination we understood the implementation of both methods and found the failure's cause.

# 5 Summary and Next Steps

A current study [5] reveals that developers ask reachability questions, which are ”searches across feasible paths through a program”. The authors show that developers often failed to understand program behavior and modified the code relying on false assumptions. Especially, the lack of adequate tool support was a reason for the developers' problems with answering reachability questions.

Dynamic service analysis [7] and our Path tool suite enable developers to experience a feeling of immediacy when they explore program behavior. Run-time views support exploring run-time behavior and facilitate answering reachability questions such as: In what context is a particular method used or where is the cause of a failure? Our approach encourages frequent use of our tools and thus promotes the validation of assumptions rather than relying on guess work.

Besides an empirical evaluation for program comprehension in the near future, we are investigating how the concepts of dynamic service analysis can facilitate answers for behavioral reachability questions. In particular, we expect benefits by extending our PathFinder to a lightweight back-in time debugger and by summarizing sane behavior within dynamic contract layers [3]. For instance, we could debug test cases backwards in time, answer *why* questions, or verify generated assertions at run-time.

# References

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

[2] O. Greevy. *Enriching Reverse Engineering with Feature Analysis*. PhD thesis, University of Berne, 2007.

[3] R. Hirschfeld, M. Perscheid, C. Schubert, and M. Appeltauer. Dynamic Contract Layers. In *SAC '10: Proceedings of the 25th Symposium on Applied Computing*, pages 2169–2175. ACM, March 2010.

[4] A. J. Ko, R. DeLine, and G. Venolia. Information Needs in Collocated Software Development Teams. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 344–353. IEEE Computer Society, 2007.

[5] T.D. LaToza and B.A. Myers. Developers Ask Reachability Questions. In *ICSE '10: Proceedings of the 32nd International Conference on Software Engineering*, pages 185–194. ACM, 2010.

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

[7] M. 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, May 2010.

[8] M. Perscheid, B. Steinert, R. Hirschfeld, F. Geller, and M. Haupt. Immediacy through Interactivity: Online Analysis of Run-time Behavior. In *WCRE '10: Proceedings of the 17th Working Conference on Reverse Engineering*, page to appear. IEEE, October 2010.

[9] 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), 2008.

[10] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold. Lightweight Fault-Localization Using Multiple Coverage Types. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, pages 56–66. IEEE, 2009.

[11] J. Sillito, G. C. Murphy, and K. De Volder. Asking and Answering Questions during a Programming Change Task. *IEEE Transactions on Software Engineering*, 34(4):434–451, 2008.

[12] B. Steinert, M. Haupt, R. Krahn, and R. Hirschfeld. Continuous Selective Testing. In *XP '10: Agile Processes in Software Engineering and Extreme Programming*, pages 132–146. Springer, 2010.

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

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