Studying the Advancement in Debugging Practice of Professional Software Developers

Benjamin Siegmund
Hasso Plattner Institute
University Potsdam
Potsdam, Germany
benjamin.siegmund@student.hpi.de

Michael Perscheid
SAP Innovation Center
Potsdam, Germany
michael.perscheid@sap.com

Marcel Taeumel
Hasso Plattner Institute
University Potsdam
Potsdam, Germany
marcel.taeumel@hpi.de

Robert Hirschfeld
Hasso Plattner Institute
University Potsdam
Potsdam, Germany
robert.hirschfeld@hpi.de

***Abstract*—In 1997, Henry Lieberman stated that debugging is the dirty little secret of computer science. Since then, several promising debugging technologies have been developed such as back-in-time debuggers and automatic fault localization methods. However, the last study about the state-of-the-art in debugging is still more than 15 years old and so it is not clear whether these new approaches have been applied in practice or not.**

For that reason, we investigate the current state of debugging in a new comprehensive study. First, we review the available literature and learn about current approaches and study results. Second, we observe several professional developers while debugging and interview them about their experiences. Based on these results, we create a questionnaire that should serve as the basis for a large-scale online debugging survey later on. With these results, we expect new insights into debugging practice that help to suggest new directions for future research.

I. Introduction

"Debugging is twice as hard as writing the program in the first place" [1]. This quote of Brian W. Kerningham illustrates a problem every software developer has to face. Debugging software is difficult and, therefore, takes a long time, often more than creating it [2]. When debugging, developers have to find a way to relate an observable failure to the causing defect in the source code. While this is easy to say, the distance from defect to failure may be long in both time and space. Developers need a deep understanding of the software system and its environment to be able to follow the infection chain back to its root cause. While modern debuggers can aid developers in gathering information about the system, they can not relieve them of the selection of relevant information and the reasoning. Debugging remains a challenging task demanding much time and effort.

Several researchers, educators, and experienced professionals have tried to improve our understanding of the knowledge and activities included in debugging programs. The earliest studies trying to understand how debugging works date as far back as 1974 [3]. In the following years, debugging tools have been improved and the lack of knowledge has been tackled [2], [4]–[6]. However, in 1997 Henry Liebermann had to say that "Debugging is still, as it was 30 years ago, largely a matter of trial and error." [7]. Also, a more recent survey from 2008 still indicates that debugging is seen as problematic and inefficient in professional context as ever [8]. The main reasons seem to be aged debugging tools and a lack of knowledge of modern debugging methods. Since that time, researchers proposed still more advanced debugging tools and methods. For example, back-in-time debuggers [9] that allow developers to follow infection chains back to their root causes or multiple automatic fault localization methods [10] that automatically highlight faulty statements in programs. Nevertheless, so far it is not clear whether the current advancement in research has already improved the situation in practice or not. For that reason, we state our research question as follows:

> Have professional software developers changed their way of debugging by using recent achievements in debugging technology?

We aim to answer this question by studying debugging in the field—observing the number of bugs, the time of detection, and the effort to fix them. First, we started with a comprehensive literature review that revealed current debugging trends and existing study results. After that, we visited four software companies in Germany and interviewed a total of eight developers. With these results, we got first insights into current debugging practices and derived an online questionnaire that is currently being answered. The contributions of this paper are:

- Review of the available literature on studies on debugging behaviour
- A field study in four companies with eight developers in order to learn about their debugging habits
- Deriving a questionnaire for a larger debugging survey

The next section gives more details on our literature review. Section III explains the design of our field study and Section IV presents its results. We address some threats to the validity in Section V. In Section VI, we describe the questions of our online survey. Finally, we conclude in Section VII.

II. Literature Review

The earliest study of debugging behaviour dates as far back as 1974. Gould and Drongowski [3] conducted a lab study with 30 participants. Each participant was given a printed Fortran source code and varying additional information. They then had to find an artificially inserted bug in that source code. Due to the limitations of the time, the developers could not execute the program. Nevertheless, the authors observed similarities amongst all developers: they scanned the code

for usual suspects before trying to actually understand its behaviour.

A year later, the same authors studied developers equipped with a symbolic debugger [11]. While the use of the debugger did not improve but lengthen the debugging times, this was attributed to distorting factors. It was only used for bugs that were to hard to solve without and the programs used were short and with a linear flow of control. The authors formulated a "gross descriptive model of debugging" which consisted of an repeated iteration of three steps:

1) Select a debugging tactic
2) Try to find a clue
3) Generate a hypothesis based on the clue if any

This was also the first time that evidence had been found for backwards reasoning from observable failure to root cause.

In 1982, Weiser introduced the notion of program slicing [12]. Developers debugged one of three Algol-W programs that contained an artificially inserted bug and were afterwards asked to identify statements taken from that program. The results showed that programmers could remember statements that influenced or were influenced by the statement containing the bug better than unrelated statements.

In 1985, Vessey found evidence that programming experts and novices differ in their debugging strategies [13]. She found out that experts are more flexible in choosing their tactics and develop an overall program understanding. Novices who lack that understanding are often constrained by their initial tactic and hypothesis, even if both turned out to be not useful.

In 1997, Eisenstadt [14] collected 59 bug anecdotes from experts and proposed a three dimensional classification of bug stories:

1) The reason, why the bug was difficult.
2) The type of the root cause identified.
3) The most useful technique used to find the root cause.

He then identified two main sources for difficult bugs: large gaps between root cause and failure and bugs that render tools inapplicable. The results also showed that these bugs can be solved by gathering and examining additional runtime data.

Since 2000, many researchers have tried to improve the understanding of specific aspects of debugging [9], [15]–[22], but these focus mostly on the introduction of new tools or how debugging can be taught to students. For example, there have been multiple approaches to automate parts of the fault localization process [23]–[40], a categorization and overview can be found in [10]. Moreover, a complete discussion of current debugging approaches can be found in [41].

Many software developers have also tried to create a guideline that can help other troubled developers improve their debugging skills and reduce time and effort spent on debugging in favour of developing new features. Some of these discussions resulted in debugging guides published as books.

"Debugging: The 9 Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems" [6] was one of the first general purpose debugging books. It teaches general strategies and how to apply them to real life bug stories by example.

"Debugging by Thinking" [5] relates debugging with other domains of problem solving and tries to apply their methods. It also provides a list of debugging strategies, heuristics and tactics, each with a detailed instruction how and when it can be applied.

"The Developer's Guide to Debugging" [4] teaches techniques to solve specific types of problems, that are usually very challenging. It exemplifies them using GDB, Visual Studio and other Tools applicable to C and C++.

"Why Programs Fail" [2] introduces the reader to the infection chain and how the knowledge of its existence can help in debugging and bug prevention. It teaches formal processes for testing, problem reproduction, problem simplification and actual debugging. It promotes *Scientific Debugging*, a debugging method based on the scientific method of generating theories. It involves repeatedly formulating hypotheses, planning and executing experiments for verification, and refining hypotheses until the root cause is found.

In the last years, we have seen many new debugging tools and methods. However, to the best of our knowledge, the latest general purpose debugging study amongst professional software developers is more than 15 years old [14]. For that reason, we argue that it is necessary to update our knowledge about professional software debugging to know which problems are still open and which should be solved next.

III. Experimental Setup of the Field Study

Goal of the field study was getting an impression of professional debugging in modern software companies. This impression was necessary to create a general debugging questionnaire. We visited four software companies in Germany varying in size from five to several hundred employees. All four companies are creating web applications, some self hosted and some licensed. We could follow eight developers through the course of their day and observe their methods. We asked each developer to *think aloud* so we could get an impression of their methods. At the end of each visit, we asked each developer to describe his overall process himself. We also asked if they knew modern tools such as back-in-time debuggers and if they deemed them useful.

An overview of the relevant characteristics of each company is given in table I. For each company it shows the number of employees, the number of software developers, the number of developers we observed, and the usual size of teams in that company, as well as the development process they used, the technology they built upon, and the tools they applied. These lists are not exhaustive but rather show the tools and technologies we could see during our visits. In addition to that data, it is worth noting that the third company is part of a larger Web-oriented enterprise.

Table II shows an overview of the relevant characteristics of the individual participants. We asked for their age, highest educational degree, and experience in software development. We also noted their gender and current position.

TABLE I. RELEVANT CHARACTERISTICS OF THE FOUR COMPANIES VISITED IN THE FIELD STUDY

	# Employees	# Developers	# Observed	Team Size	Process	Used Technologies	Used Tools
A	300	50	3	7	Scrum 3 week sprints	Java EE, Hibernate, JUnit, JSF, ANT, JBoss, Tomcat	Jira, Jenkins, Git, Eclipse, PL/SQL Developer
B	25	15	2	5	Kanban	Java, ANT, JUnit, Tomcat, XML, SVG, JavaScript, NodeJS, Grunt, Jasmine, Karma	Jira, Jenkins, Git, Eclipse, Sublime Text, Chrome DevTools
C	150	60	1	5	Kanban	Java EE, ANT, Sonar, Tomcat, Morphia, JSON, MongoDB	Jira, Jenkins, Git, Eclipse
D	5	3	2	5	Scrum weekly sprints	PHP, Zend, Propel, MySQL, New Relic, JavaScript, XHTML, JQuery	Jira, Hudson, Git, Sublime Text, Chrome DevTools, PHPStorm, Apache

TABLE II. RELEVANT CHARACTERISTICS OF THE PARTICIPANTS OF THE FIELD STUDY

Company	Age	Gender	Degree	Experience	Position
A	40	male	Diploma in Engineering	8 years freelance web development 3 years web front-end development 1 year back-end development	Java back-end developer
A	26	male	Master in Computer Science	2 years back-end development	Java back-end developer
A	31	male	Bachelor in Computer Science	5 years back-end development	Java back-end developer
B	27	male	Master in IT-Systems-Engineering	6 years miscellaneous 1 year JavaScript development	developing a JavaScript graphics library
B	28	male	Master in Engineering	2 years back-end development	Java back-end developer
C	30	male	Master in Computer Science	7 years back-end development	Java back-end developer
D	27	male	Bachelor in Artificial Intelligence and Computer Science	4 years front-end development	Web front-end developer
D	34	male	Bachelor in Computer Science Certified IT-Specialist	15 years miscellaneous one year back-end development	PHP back-end developer

IV. STUDY RESULTS

A. Company A

The development process of the first company includes a mandatory code review for each feature or fix. Each team had a dedicated quality assurance employee, who performs manual and automated integration and acceptance tests. They also regularly execute automated unit tests written by the developers themselves.

The first developer we observed uses full-text search and the search-for-class utilities of the Eclipse IDE to navigate the source code. When confronted with unexpected behaviour, he first checks which code was recently modified and therefore might probably contain the fault. He then sets breakpoints at key locations of the program flow to interrupt the program and check the program state. Checking database contents required a separate tool. Interrupting the program to check its state is also a preventive instrument to him, when new code is complex or uses unfamiliar interfaces. He is not aware of any standard approach, but his approach can be classified as scientific debugging [2] without taking notes, as he formulates hypotheses and then checks these by experiment.

The second developers' source code navigation methods of choice are the search-for-class, jump-to-implementation, and find-callers utilities of the Eclipse IDE. When debugging, he makes sure to work on the exact same Git branch the bug was found on to eliminate possible version dependencies. He then inspects the latest changes on that branch utilizing the capabilities of git to show differences between commits. Explaining his approach is as hard to him as to the first developer, but he also follows a simple version of scientific debugging, setting breakpoints to inspect the program state to verify assumptions. He calls this an "intuitive method". When testing hypotheses the hot recompile capabilities of Java proved allowed him to change the code at runtime and proceed in a trial and error fashion until understanding the problem.

When we visited the company, the third developer had to find the cause of a dependency conflict. A class included in multiple libraries was delivered in different, not compatible versions. To find out which jar files included the class, he first inspected the state of the Java Runtime Environment using print statements to get a list of all jar files actually loaded. He then used the command line tool `grep` to check the content of these files for the conflicting class. After spending a reasonable amount of time and effort this way, he postponed the fix and planned to improve the overall dependency management instead. This also meant postponing tasks depending on a new library that introduced the conflicts.

B. Company B

The development process of the second company includes a mandatory code review for each feature or fix. Following test-driven development, the general process for new features includes an automated "happy case" test written upfront and automated edge case tests written after or while implementing. Bug reports were created on GitHub, either by a customer or by support employees. If the bug is simple, support employees fix it themselves, but most problems are only reproduced by support and fixed by developers. Some cases cannot be reproduced because of third party systems the customer uses. In that case, support employees try to help with diagnosis until the problem is either solved or can be reproduced using substitutes.

The first participant in this company uses mainly full text

search or a search for symbols provided by Sublime Text to navigate the code. New automated test cases mark the beginning of each of his debugging sessions. At this point, he usually has a first hypothesis, that can be tested by setting breakpoints at relevant locations and inspecting the program state. To gather more data, he uses the interactive console of the Google Chrome development tools to explore objects and APIs. When examining the program flow, he makes extensive use of stepping and the "restart frame" functionality of the Google Chrome debugger. At one instance, he refactored the program and ran his test suite again to verify his internal model of the program.

The other developers' source code navigation tools are full text search and many of the navigation utilities provided by the Eclipse IDE. He follows a simple debugging philosophy called "Test it, don't guess it", which can be seen as a simplified version of scientific debugging. When confronted with a runtime exception he reads the stack trace provided very carefully to identify the relevant classes and methods, proceeding by setting breakpoints, stepping through the program, and inspecting the program state to verify hypotheses. When needing backwards navigation he uses Eclipses "Drop to Frame" utility where applicable.

C. Company C

The development process of the third company includes mandatory code review for each new feature or fix. There is a separate quality assurance department that performs automated as well as manual testing. Unit tests written by the developers themselves complement the test suite.

We observed only one developer in this company. To navigate the source code, he uses full-text search as well as the navigation utilities provided by Eclipse. His general debugging approach usually starts at the beginning of a relevant use case, stepping into the program and inspecting variables to get an impression of the program and data flow. He then starts setting breakpoints at relevant locations and testing hypotheses. Exception breakpoints, capturing all exceptions, even if caught by the program, provide him with further data. When inspecting complex objects, he sometimes writes a custom `toString()` method to aid the investigation.

D. Company D

The development process of the fourth company includes an optional code review and automated unit tests. The review is not mandatory because they deem the slow down too heavy for a start-up needing to evolve quickly. A beta tester group of users reports bugs unnoticed by the developers themselves.

The front-end developer uses only full-text search and search for files by name to navigate the source code. When debugging, his first step is reading recent source code, checking it for obvious mistakes. He then uses the Google Chrome debugger to set breakpoints, step through the program and inspect variables, using the interactive console to explore objects and APIs. When working with the (X)HTML document, he uses the inspector to examine the results of the code.

The back-end developers' code navigation tools are full-text search, manual folder navigation and the "find implementers" and "find callers" utilities of the PHPStorm IDE. When working on a bug report, he first examines the logs of the New Relic monitoring system to get a impression of the system parts involved, proceeding by setting breakpoints and examining the program state and flow to test hypotheses. He also compares the defective modules to working ones which employ the same patterns and use the same APIs to check for differences.

E. General Findings

While the level of education and amount of practical experience varies among the developers, all reported that they were never trained in debugging. They learned debugging either by doing or from demonstration by colleagues. Not surprisingly, they have difficulties describing their approach. While they can speak about development processes in general on an abstract level, they resort to showing and examples when speaking about debugging.

All developers use a simplified scientific method, although they did not describe it using that name. They formulate hypotheses about the program and then set up simple experiments to verify them. They do neither take notes nor mark their results in the source code, though. This might be a hint that scientific debugging is a way of thought that comes easy to most developers.

Some developers allowed us to see how they formed their initial hypotheses. They use stack traces, log files and review the code to identify related modules, classes and methods. They then use their system knowledge and reference material to identify suspicious code in these parts of the program. Others were able to formulate an initial hypothesis just after reading the bug report. This is probably due to different levels of difficulty of the bugs they encountered and also their knowledge about the system.

All participants are proficient in using symbolic debuggers. They also prefer them to the use of log statements, because debuggers allow for additional inspections without the need to restart the program. Only a few developers claim that they are aware of all features of their IDE or debugger, though. Unknown features include "Drop to Frame" or "Restart Frame", conditional breakpoints, and various kinds of special breakpoints.

No subject had known back-in-time debuggers before. All of them question the usefulness of back-stepping by deeming it sufficient to set a breakpoint earlier in the program and rerun the test. A back-in-time debugger is only considered useful if it has only a very small overhead and memory footprint when compared to a regular debugger. Automatic fault localization was also unknown, but the suspects deem it more useful, depending on the analysis runtime and difficulty of the debugging session. They consider running an analysis overnight to localize a bug that could not be found till the end of a workday a viable option.

V. Threats to Validity

The results of this field study can not be generalized. The main concern is the small scale. Eight developers are not enough to rule out statistical anomalies, the same goes for four companies. Another concern is the limited time span.

Each developer was only observed for some hours during one workday. This results in a limited sample of problems they might encounter during day to day work limiting the observed methods and approaches. Furthermore all companies created web applications, which may or may not result in a similar company culture. Nevertheless, these interviews provide meaningful insights that helped us design our debugging questionnaire.

VI. Preparing an Online Questionnaire

To consolidate and expand our results with reliable statistic data we are performing an online survey. Based on the results mentioned in section IV, we formulated 34 questions on debugging tools, workload, approach and education as well as bug tracking and bug prevention. At the time of writing, the online survey as well as a printable version are available at https://www.uni-potsdam.de/skopie-up/index.php/689349.

At the beginning of the survey we collect some background information, namely age, gender, education, development experience, the size of the companies the participants work for, and the programming languages they use. This information will help us to identify influencing factors for the approach developers have when debugging software.

Because modern tools are largely unknown to the participants in the field study, the questionnaire asks which tools the developer knows and which he uses for debugging. We divide debugging tools in 13 categories: printing and logging, assertions and Design by Contract, symbolic debuggers, back in time debuggers, likely invariants, program slicing, slice based fault localization, spectrum based fault localization, statistics based fault localization, program state based fault localization, machine learning based fault localization, model based fault localization, and data mining based fault localization. The distinction of the automatic fault localization methods is taken from Wong [10]. We also ask how much developers value different aspects of new debugging tools. The available aspects are features, overhead or runtime, IDE integration, easy installation, easy to use, and available documentation or support.

We then try to assess the participants' debugging workload by asking how much of their time they spent developing and how many bugs of different difficulties they encounter. We also ask them to estimate if the difficulty of debugging has changed in the last years or will change in the next.

As difficult bugs often enable deeper insights into the developers approach, we include some questions on the hardest bug the participants had to face. We ask them to position it in all three dimensions of a bug war story Eisenstadt [14] defined: The type of the bug, why it was especially hard to debug, and what technique turned out to be most helpful to find it. Early tests have shown that many of the bugs remembered as the hardest are due to parallel execution and do not fit in the existing categories. Therefore, we added the category of parallel problem to the root cause dimension. We also ask how long it took to fix the bug.

Because the participants of the field study show difficulties explaining their approach, we want to find out if that is a common problem. To this end, we ask participants of the online survey to describe their standard approach, if they follow one. As our field study indicates written records being uncommon, we also ask if the participants keep such and what kind of records they use.

Zeller [2] has devoted a whole chapter on learning from past bugs. He explains how to utilize the bug history of a project to identify problems in the development process and to fix future bugs faster. Therefore, we ask the participants if they keep a log of fixed bugs, if they add solutions to their log and if they use it to learn from past mistakes.

We also want to analyse bug prevention and its effects on the remaining bugs. Therefore, we include questions in the survey to assess what type of automated tests and analysis the participants perform.

Lastly, we are interested in the influence of education on debugging. To this end, we ask the participants if they got any debugging education and when that was. We also ask if they have read any literature on debugging methods.

With all these questions we hope to find out to what extent the advancements in the debugging technology have entered the field of professional software development. We want to assess what factors influence the adoption of new tools or methods and what directions future research and education should take to further improve developers' debugging abilities.

VII. Conclusion

In this paper, we presented our results of studying debugging behaviour of professional software developers. We reviewed the available literature and noted a 17 year gap since the last comparable study. We performed an explorative field study, visiting four companies in Germany and observing a total of eight developers in their habitual working environment. All of them were proficient in using a symbolic debugger. Although all followed a standard approach that can be seen as a simplified scientific method, none of them was aware of this or able to explain his approach without resorting to demonstration. None of them had any formal education in debugging and also nobody had knowledge of back in time debuggers or automatic fault localisation techniques. Based on these results, we created an online survey to consolidate and expand our results.

References

[1] B. W. Kernighan and P. J. Plauger, "The elements of programming style," *McGraw-Hill*, vol. 1, 1978.

[2] A. Zeller, *Why programs fail: a guide to systematic debugging*. Morgan Kaufmann, 2009.

[3] J. D. Gould and P. Drongowski, "An exploratory study of computer program debugging," *The Journal of the Human Factors and Ergonomics Society*, vol. 16, no. 3, pp. 258–277, 1974.

[4] T. Grötker, U. Holtmann, H. Keding, and M. Wloka, *The developer's guide to debugging*, 2nd ed. Self-publishing company, 2012.

[5] R. C. Metzger, *Debugging by thinking: A multidisciplinary approach*. Elsevier Digital Press, 2004.

[6] D. J. Agans, *Debugging: The 9 indispensable rules for finding even the most elusive software and hardware problems*. AMACOM Div American Mgmt Assn, 2002.

[7] H. Lieberman, "The debugging scandal and what to do about it (introduction to the special section)," *Commun. ACM*, vol. 40, no. 4, pp. 26–29, 1997.

[8] M.-C. Ballou, "Improving software quality to drive business agility," *IDC Survey and White Paper*, 2008.

[9] B. Lewis, "Debugging Backwards in Time," in *Proceedings of the International Workshop on Automated Debugging*, ser. AADEBUG. Arxiv, 2003, pp. 225–235.

[10] W. E. Wong and V. Debroy, "A survey of software fault localization," *Department of Computer Science, University of Texas at Dallas, Tech. Rep. UTDCS-45-09*, 2009.

[11] J. D. Gould, "Some psychological evidence on how people debug computer programs," *International Journal of Man-Machine Studies*, vol. 7, no. 2, pp. 151–182, 1975.

[12] M. Weiser, "Programmers use slices when debugging," *Communications of the ACM*, vol. 25, no. 7, pp. 446–452, 1982.

[13] I. Vessey, "Expertise in debugging computer programs: A process analysis," *International Journal of Man-Machine Studies*, vol. 23, no. 5, pp. 459–494, 1985.

[14] M. Eisenstadt, "My hairiest bug war stories," *Communications of the ACM*, vol. 40, no. 4, pp. 30–37, 1997.

[15] R. Lencevicius, "On-the-fly query-based debugging with examples," *arXiv*, 2000.

[16] B. Hailpern and P. Santhanam, "Software debugging, testing, and verification," *IBM Systems Journal*, vol. 41, no. 1, pp. 4–12, 2002.

[17] R. Chmiel and M. C. Loui, "Debugging: from novice to expert," in *ACM SIGCSE Bulletin*, vol. 36, no. 1. ACM, 2004, pp. 17–21.

[18] M. Ahmadzadeh, D. Elliman, and C. Higgins, "An analysis of patterns of debugging among novice computer science students," in *ACM SIGCSE Bulletin*, vol. 37, no. 3. ACM, 2005, pp. 84–88.

[19] S. James, M. Bidgoli, and J. Hansen, "Why sally and joey can't debug: next generation tools and the perils they pose," *Journal of Computing Sciences in Colleges*, vol. 24, no. 1, pp. 27–35, 2008.

[20] L. Murphy, G. Lewandowski, R. McCauley, B. Simon, L. Thomas, and C. Zander, "Debugging: the good, the bad, and the quirky–a qualitative analysis of novices' strategies," *ACM SIGCSE Bulletin*, vol. 40, no. 1, pp. 163–167, 2008.

[21] B. Hanks and M. Brandt, "Successful and unsuccessful problem solving approaches of novice programmers," *ACM SIGCSE Bulletin*, vol. 41, no. 1, pp. 24–28, 2009.

[22] T. D. LaToza and B. A. Myers, "Developers ask reachability questions," in *International Conference on Software Engineering*, vol. 1. IEEE, 2010, pp. 185–194.

[23] H. Agrawal, J. Horgan, S. London, and W. Wong, "Fault localization using execution slices and dataflow tests," *Conference on Software Reliability Engineering*, pp. 143–151, 1995.

[24] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Practical fault localization for dynamic web applications," in *International Conference on Software Engineering*. ACM, 2010, pp. 265–274.

[25] P. Arumuga Nainar and B. Liblit, "Adaptive bug isolation," in *International Conference on Software Engineering*. ACM, 2010, pp. 255–264.

[26] G. K. Baah, A. Podgurski, and M. J. Harrold, "Causal inference for statistical fault localization," in *International Symposium on Software Testing and Analysis*. ACM, 2010, pp. 73–84.

[27] H. Cleve and A. Zeller, "Locating causes of program failures," in *International Conference on Software Engineering*. ACM, 2005, pp. 342–351.

[28] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight defect localization for java," in *European Conference on Object-Oriented Programming*. Springer, 2005, pp. 528–550.

[29] N. Gupta, H. He, X. Zhang, and R. Gupta, "Locating faulty code using failure-inducing chops," in *International Conference on Automated Software Engineering*. ACM, 2005, pp. 263–272.

[30] T. Janssen, R. Abreu, and A. J. van Gemund, "Zoltar: A toolset for automatic fault localization," in *International Conference on Automated Software Engineering*. IEEE Computer Society, 2009, pp. 662–664.

[31] D. Jeffrey, N. Gupta, and R. Gupta, "Fault localization using value replacement," in *International Symposium on Software Testing and Analysis*. ACM, 2008, pp. 167–178.

[32] L. Jiang and Z. Su, "Context-aware statistical debugging: from bug predictors to faulty control flow paths," in *International Conference on Automated Software Engineering*. ACM, 2007, pp. 184–193.

[33] J. A. Jones, J. F. Bowring, and M. J. Harrold, "Debugging in parallel," in *International Symposium on Software Testing and Analysis*. ACM, 2007, pp. 16–26.

[34] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *ACM SIGPLAN Notices*, vol. 40, no. 6. ACM, 2005, pp. 15–26.

[35] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "Sober: statistical model-based bug localization," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 286–295, 2005.

[36] S. Park, R. W. Vuduc, and M. J. Harrold, "Falcon: fault localization in concurrent programs," in *International Conference on Software Engineering*. ACM, 2010, pp. 245–254.

[37] M. Renieres and S. P. Reiss, "Fault localization with nearest neighbor queries," in *International Conference on Automated Software Engineering*. IEEE, 2003, pp. 30–39.

[38] C. Yilmaz, A. Paradkar, and C. Williams, "Time will tell: fault localization using time spectra," in *International Conference on Software Engineering*. ACM, 2008, pp. 81–90.

[39] A. Zeller, "Isolating cause-effect chains from computer programs," in *Symposium on Foundations of Software Engineering*. ACM, 2002, pp. 1–10.

[40] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *International Conference on Software Engineering*. ACM, 2006, pp. 272–281.

[41] M. Perscheid, "Test-driven fault navigation for debugging reproducible failures," Ph.D. dissertation, Hasso Plattner Institute, University of Potsdam, 2013.