Software Qual J (2017) 25:83–110
DOI 10.1007/s11219-015-9294-2

Studying the advancement in debugging practice of professional software developers

Michael Perscheid[1] · Benjamin Siegmund[2] · Marcel Taeumel[2] · Robert Hirschfeld[2]

Published online: 11 January 2016

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 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. Third, we create a questionnaire that serves as the basis for a larger online debugging survey. Based on these results, we present new insights into debugging practice that help to suggest new directions for future research.

Keywords Debugging · Literature review · Field study · Online survey

1 Introduction

"Debugging is twice as hard as writing the program in the first place" (Kernighan and Plauger 1978). This quote of Brian W. Kernighan illustrates a problem every software developer has to face. Debugging software is difficult and, therefore, takes a long time, often more than creating it (Zeller 2009). 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

✉ Michael Perscheid
michael.perscheid@sap.com

Robert Hirschfeld
robert.hirschfeld@hpi.de

[1] SAP Innovation Center, Potsdam, Germany

[2] Hasso Plattner Institute, University of Potsdam, Potsdam, Germany

cause. While modern debuggers can aid developers in gathering information about the system, they cannot 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 (Gould and Drongowski 1974). In the following years, debugging tools have been improved and the lack of knowledge has been tackled. However, more than 20 years later, Henry Lieberman had to say that "Debugging is still, as it was 30 years ago, largely a matter of trial and error." (Lieberman 1997). Also, a more recent survey from 2008 still indicates that debugging is seen as problematic and inefficient in professional context as ever (Ballou 2008). 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 (Zeller 2009). For example, back-in-time debuggers (Lewis 2003) that allow developers to follow infection chains back to their root causes or multiple automatic fault localization methods (Wong and Debroy 2009) 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 a systematic questionnaire that has been answered in a larger online survey. The contributions of this paper are:

- Review of the available literature on studies on debugging behavior
- A field study in four companies with eight developers in order to learn about their debugging habits
- Deriving a questionnaire and conducting an online survey in order to reveal current debugging practices and propose further research directions

Compared to the previous version of this paper as presented at the Fifth International Workshop on Program Debugging (2014), we include the results of our online survey and infer promising directions for the future of debugging.

The remainder of this paper is structured as follows: Section 2 gives more details on our literature review. Section 3 explains the design of our field study and presents its results. Section 4 describes the questions of our online survey, discusses the answers, and proposes possible research directions. Section 5 concludes.

2 Literature review

The earliest study of debugging behavior dates as far back as 1974. Gould and Drongowski (1974) conducted a laboratory 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 among all developers: they scanned the code for usual suspects before trying to actually understand its behavior.

A year later, the same authors studied developers equipped with a symbolic debugger (Gould 1975). 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 a 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 (Weiser 1982). Developers debugged one of three Algol-W programs that contained an artificially inserted bug and were afterward 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 (Vessey 1985). 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 (1997) 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 (Ahmadzadeh et al. 2005; Chmiel and Loui 2004; Hailpern and Santhanam 2002; Hanks and Brandt 2009; James et al. 2008; LaToza and Myers 2010; Lencevicius 2000; Lewis 2003; Murphy et al. 2008), 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 (Agrawal et al. 1995; Artzi et al. 2010; Arumuga et al. 2010; Baah et al. 2010; Cleve and Zeller 2005; Dallmeier et al. 2005; Gupta et al. 2005; Janssen et al. 2009; Jeffrey et al. 2008; Jiang and Su 2007; Jones et al. 2007; Liblit et al. 2005; Liu et al. 2005; Park et al. 2010; Renieres and Reiss 2003; Yilmaz et al. 2008; Zeller 2002; Zhang et al. 2006), a categorization and overview can be found in Wong and Debroy (2009). Moreover, a complete discussion of current debugging approaches can be found in Perscheid (2013).

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 favor 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" (Agans 2002) 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" (Metzger 2004) 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" (Grötker et al. 2012) 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" (Zeller 2009) 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 among professional software developers is more than 15 years old (Eisenstadt 1997). 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.

3 Field study

This section explains the setup and results of our field study. The goal was to get an impression of professional debugging in modern software companies. This impression was necessary to decide which questions might produce interesting insights. To get that impression, we decided to visit companies and observe developers during a normal workday. We contacted many companies on a university relation fair and were able to convince four of them to help us.

3.1 Experimental setup

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 Tables 1 and 2. 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

Table 1 Relevant characteristics of the four companies visited in the field study

	# Employees	# Developers	# Observed	Team size	Process
A	300	50	3	7	Scrum 3 week sprints
B	25	15	2	5	Kanban
C	150	60	1	5	Kanban
D	5	3	2	5	Scrum weekly sprints

Table 2 Used technologies and tools of the four visited companies

	Used technologies	Used tools
A	Java EE, Hibernate, JUnit, JSF, ANT, JBoss, Tomcat	Jira, Jenkins, Git, Eclipse, PL/SQL Developer
B	Java, ANT, JUnit, Tomcat, XML, SVG, JavaScript, NodeJS, Grunt, Jasmine, Karma	Jira, Jenkins, Git, Eclipse, Sublime Text, Chrome DevTools
C	Java EE, ANT, Sonar, Tomcat, Morphia, JSON, MongoDB	Jira, Jenkins, Git, Eclipse
D	PHP, Zend, Propel, MySQL, New Relic, JavaScript, XHTML, JQuery	Jira, Hudson, Git, Sublime Text, Chrome DevTools, PHPStorm, Apache

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.

Tables 3 and 4 show an overview of the relevant characteristics of the individual participants. We asked for their age, gender, and highest educational degree. We also noted their experience in software development and current position.

3.2 Study results

3.2.1 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

Table 3 Relevant characteristics of the eight participants (#) of the field study

#	Company	Age	Gender	Degree
1.	A	40	male	Diploma in Engineering
2.	A	26	male	Master in Computer Science
3.	A	31	male	Bachelor in Computer Science
4.	B	27	male	Master in IT Systems Engineering
5.	B	28	male	Master in Engineering
6.	C	30	male	Master in Computer Science
7.	D	27	male	Bachelor in Artificial Intelligence and Computer Science
8.	D	34	male	Bachelor in Computer Science
				Certified IT Specialist

Table 4 Experience and position of participants (#)

#	Experience	Position
1.	8 years freelance web development	Java back-end developer
	3 years web front-end development	
	1 year back-end development	
2.	2 years back-end development	Java back-end developer
3.	5 years back-end development	Java back-end developer
4.	6 years miscellaneous	Developing a JavaScript graphics library
	1 year JavaScript development	
5.	2 years back-end development	Java back-end developer
6.	7 years back-end development	Java back-end developer
7.	4 years front-end development	Web front-end developer
8.	15 years miscellaneous	PHP back-end developer
	1 year back-end development	

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 behavior, 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 (Zeller 2009) 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.

3.2.2 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.

3.2.3 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.

3.2.4 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.

3.3 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.

3.4 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.

4 Online survey

While our field study provided a first idea of the debugging habits in modern software companies, it is unreasonable to use it to formulate meaningful results. To achieve a useful certainty, we needed a larger sample. However, it was unfeasible to visit hundreds of developers and observe them. An easier way is using a questionnaire that can be filled in online. With such a questionnaire, we can reach a large number of people and process the results statistically. So, we can find and describe relations between different characteristics of professional software debugging. This section summarizes the questionnaire, describes the experimental setup, details our findings, and proposes further research directions.

4.1 Formulating the questionnaire

To consolidate and expand our results with reliable statistic data, we are performing an online survey. Based on the results of our field study, we formulated a broad spectrum of debugging questions:

4.1.1 Background information

At the beginning of the survey, we collect some background information, namely age, gender, degree, development experience, the size of the companies the participants work for, and the programming languages they use.

4.1.2 Education

We want to study the spread of education on debugging, because the field study indicated it as uncommon. 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.

4.1.3 Debugging tools

Because modern tools are largely unknown to the participants in the field study, the questionnaire asks which tools developers know and which they use for debugging. We divide debugging tools into 13 categories: printing and logging, assertions and Design by Contract, symbolic debuggers, back-in-time debuggers, generated 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 and Debroy (2009). 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.

4.1.4 Workload

We then try to assess the participants debugging workload by asking how much of their time they spend debugging 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.

4.1.5 Hardest bug

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 as defined by Eisenstadt (1997): The type of the bug, why it was especially hard to debug, and what technique turned out to be the 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, if fixed at all.

4.1.6 Learning from past bugs

Bug histories of a project can identify problems in the development process and help 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.

4.1.7 Bug prevention

We also want to get an idea of the relation between bug prevention, the remaining bugs, and how they are tackled. Therefore, we include questions in the survey to assess what type of automated tests and analysis the participants perform.

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.

4.2 Experimental setup

To get as many answers as possible, we made the survey available publicly and sent the link to as many people as possible using different channels. First, we sent emails to 20 different software companies which were interested after a short talk at the Connecticum fair. Second, we sent mails to different software development-related mailing lists, including C/C++, Python, Ruby, Smalltalk, git, several Linux distributions, mailing lists

of Hackerspace, and several open-source projects. We also spread the link on social media via Facebook, Google+ and Xing groups, Twitter, Reddit, and ResearchGate. Judging on the number of answers arriving after each time we spread the link, Reddit brought in the most participants. Reddit also was the medium where we received the most feedback from participants in the form of comments.

4.3 Results

In the end, our online survey has completely been answered by 303 participants.

4.3.1 Background information

The vast majority of our participants were male (287 participants, 95 %). Only four participants reported being female and twelve gave no answer. The average age of participants is 33, the median is 31. A detailed view of the age distribution can be seen in Fig. 1.

Two hundred and thirty-two participants reported having an IT-related educational degree (77 %), while 71 said they had none (23 %). Most participants had a Bachelor's (98 participants, 32 %) or Master's degree (102 participants, 34 %), while Ph.D. (20 participants, 7 %) and Job Training (12 participants, 4 %) are less common. Of those having no IT-related degree, 23 reported having some sort of school degree, 15 reported having a Bachelor's degree, 16 reported having a Master's degree, four reported having a doctoral degree, one reported having an associate degree, and eight did not report any degree. Four participants reported having a degree not fitting in the English classification.

The distribution of experience in software development is shown in Fig. 2. It can be seen that our participants range from inexperienced developers to very experienced ones.

Forty-seven Participants reported working alone (15 %). These are either freelance developers or students. Calculated over the remaining participants, the estimated company

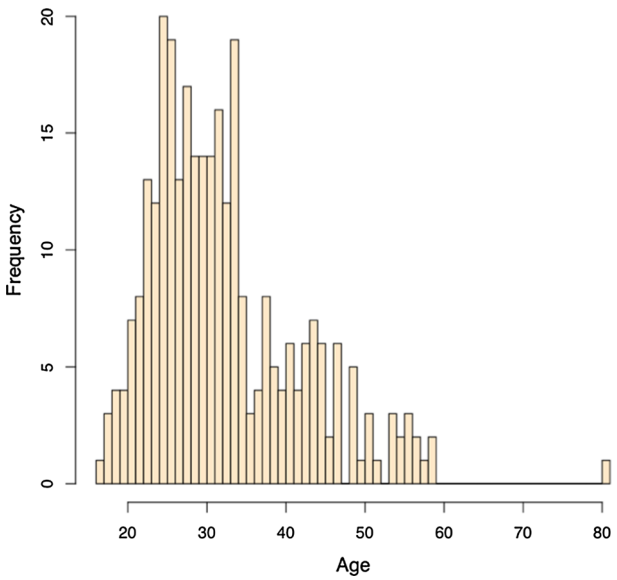

Fig. 1 How old are you?

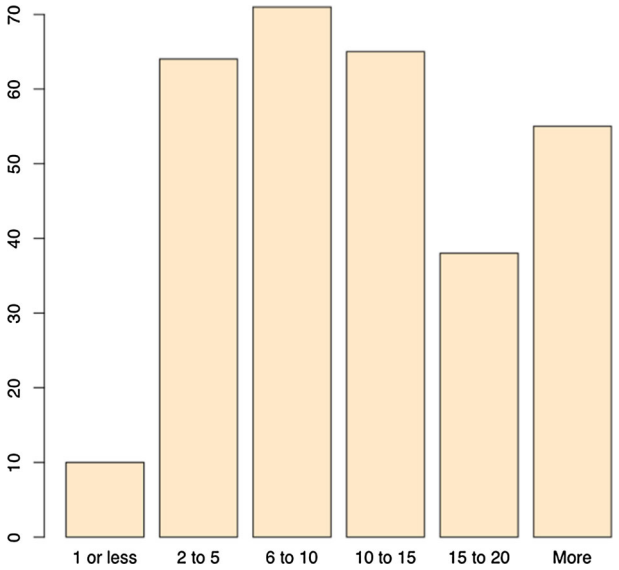

Fig. 2 For how long have you been developing software?

size ranges from two to large corporations with more than 100,000 employees. The median company size is at 200 employees. The first and third quartile are 25 and 1,400 employees. This shows that most of our participants work for small- to medium-sized companies except for some outliers. The number of software developers in the company shows a similar distribution. The maximum number of software developers is 65,000, and the median 40. The first and third quartile are eight and 250.

Finally, we asked our participants for their favorite programming language (multiple answers allowed). Figure 3 presents the results that highlight the most common answers out of 64 different programming languages in total.

All in all, we argue that *the distributions constitute a representative set for software development practice.* It reflects our observations from the field study as well as matches to our own experiences from our academic and industrial cooperation partners. We have most diverse developers from all ages, with different experiences (education as well as professional), and various programming communities.

4.3.2 Education

When asked, 156 participants said they have not received any education in debugging (51 %). In other words, *more than 50 % of our software developers had no formal debugging knowledge.* One hundred and forty-seven participants mentioned they have (49 %). Figure 4 shows where they received it. The majority of participants received their debugging education at universities or colleges but often only once. For that reason, participants having an IT-related graduation are slightly more likely to have received debugging education (Pearson coefficient 0.23[1]).

[1] The Pearson coefficient can be used to find linear relations between numeric variables. Values between −1 and 1 indicate how well the actual relation can be approximated by a linear function (Pearson 1895).

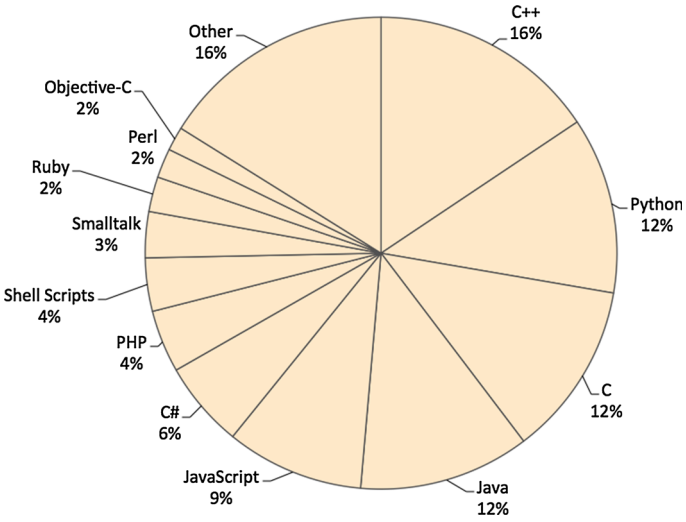

Fig. 3 What are your favorite programming languages?

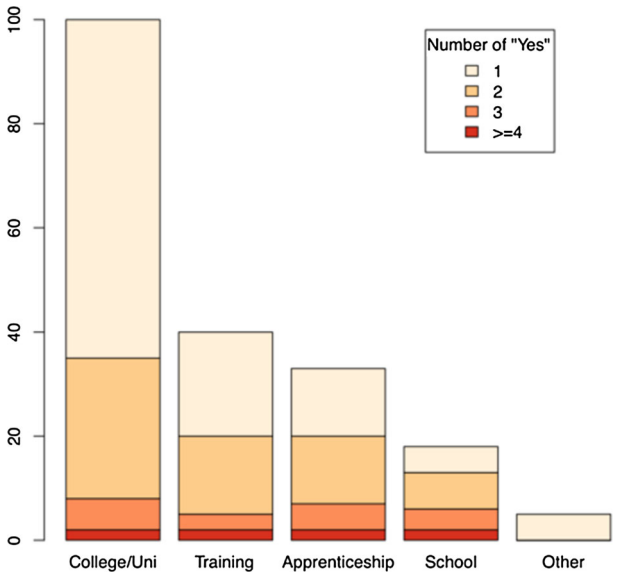

Fig. 4 Where and how often have you received debugging education?

Figure 5 shows the relation between age and debugging education. Younger participants reported having received such education more often than older participants. However, this is only a small trend and the Pearson correlation coefficient is only -0.06, indicating no linear correlation. *Both results together indicate that debugging education is still uncommon, but more university or college courses started including it recently.*

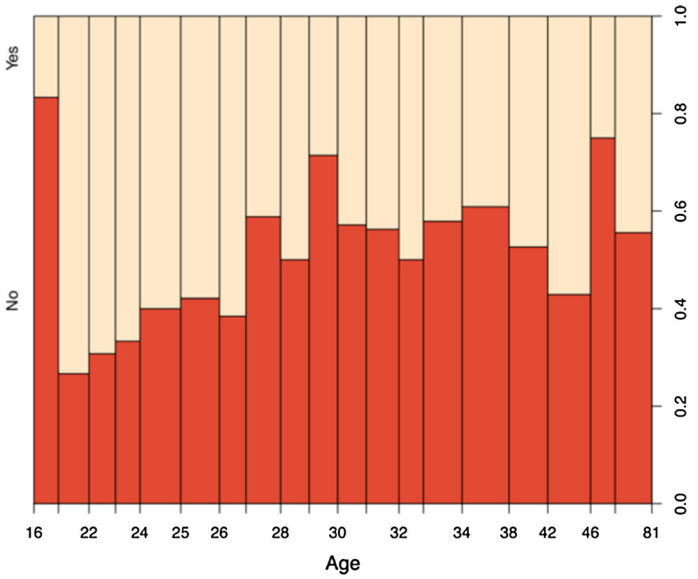

Fig. 5 Debugging education depending on age

Seventy participants reported that they never read any literature on debugging methods (23 %). The types of literature the remaining 233 read (77 %) are shown in Fig. 6, as well as if they also read other kinds. It can be seen that online media are the most common source of debugging literature. Books and scientific articles are a less common. There are only a few participants who use only one source of literature, most use two ore even more

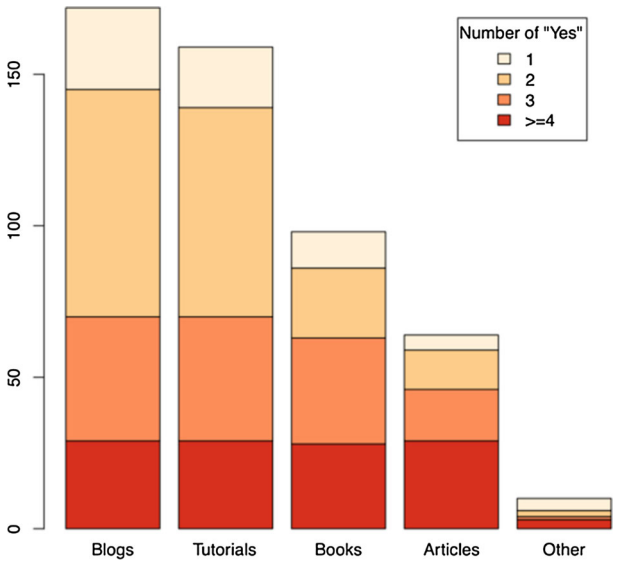

Fig. 6 What and how often have you read literature about debugging?

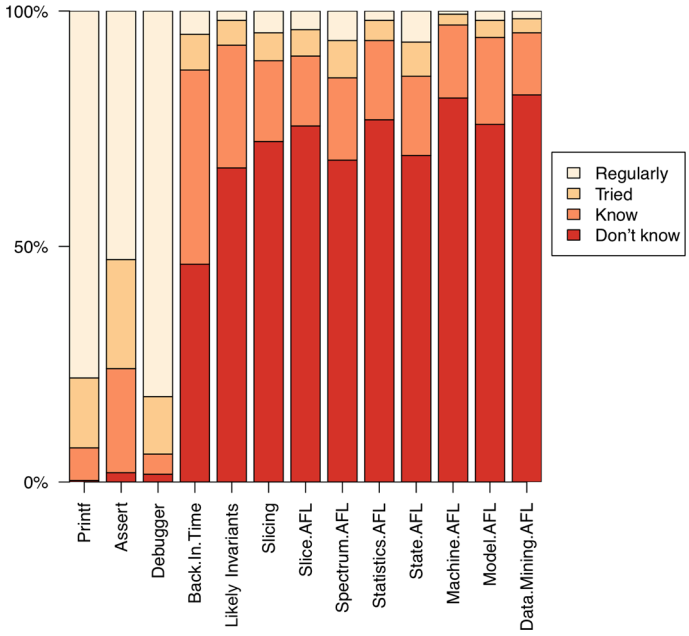

Fig. 7 Which debugging tools do you know?

types of sources. Sources mentioned as other included software documentation, magazines, the source code of debugging tools, and online lectures. *Based on these numbers, we conclude that there is a need for most developers to learn more about debugging in general.*

4.3.3 Debugging tools

As can be seen in Fig. 7, *only the older debugging tools such as printf, assertions and symbolic debuggers are commonly used by many participants.* While back-in-time debuggers are known by a larger portion, only a small number of participants use the more advanced debugging tools or Automatic Fault Localization (AFL) approaches. These observations confirm our findings from the field survey.

There are almost no correlations between the programming language used and the debugging tools used. However, participants using C++ show a small tendency to use assertions (Pearson coefficient 0.26), and participants using C show a small tendency to use printf debugging (Pearson coefficient 0.23).

To further investigate what is missing to adapt new debugging tools in practice, we asked our participants about the importance of properties new tools need to fulfill. Figure 8 shows the distribution of answers. Overall, every property is considered extremely important by some participants. *Most important to our participants are ease of use and available documentation.* Runtime overhead and an easy installation are less important than the actual debugging features. Finally, IDE Integration is most often considered not that important.

We also checked for correlations between both debugging tools questions. The highest Pearson coefficients were only 0.20, indicating that there are either no or only small

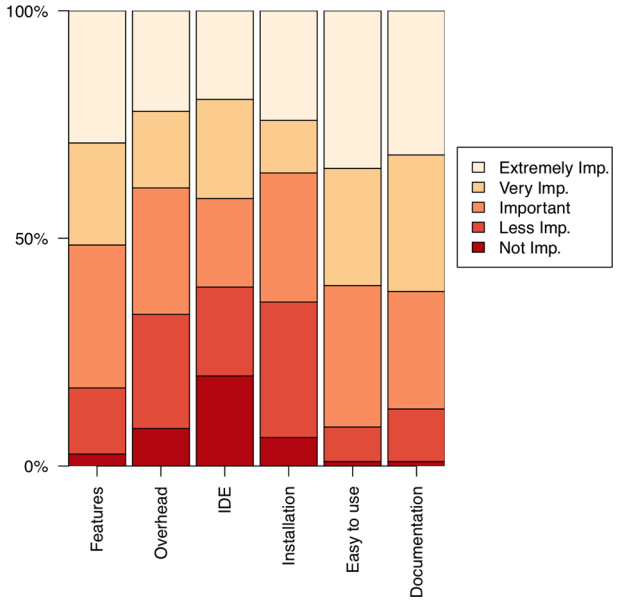

Fig. 8 What properties are important for new debugging tools?

correlations. Nevertheless, we found a small relation between participants who value a small overhead and yet apply generated likely invariants. This can imply that likely invariant tools such as Daikon seem to have a feasible runtime. However, we also found that participants who prefer rich documentation are less likely to apply back-in-time debuggers. This may indicate a lack of documentation for current available tools. There are no strong correlations between the programming language used and the properties of debugging tools participants value. Neither the use or knowledge of tools nor the properties valued correlate with participant age, company size, or debugging education.

4.3.4 Workload

Debugging is known to be an integral part of daily software development. Most developers (144 participants, 47 %) spend 20 to 40 % of their work time for debugging. The next largest groups estimate using 40–60 % (77 participants, 26 %) or 0–20 % (54 participants, 17 %). Twenty-six participants spend 60–80 % (9 %) and only two developers spend more than 80 % (1 %) of their time for debugging.

Figure 9 shows the number of bugs participants reported encountering per month. As could be expected, they report facing many easy and only a hand full hard bugs. The answers were strict descending with rising difficulty for 222 participants (73 %).

When relating work time for debugging with the number of bugs encountered per month, one would expect participants spending less time on debugging to encounter less or less difficult bugs. Using the means of the choosable intervals, the highest Pearson coefficients are 0.18 for medium and 0.16 for hard. All other coefficients indicate no correlation. *This indicates that there is no overall pattern in the difficulty of bugs that make people spend time on debugging.* However, medium and hard bugs are more related to high percentages of debugging time than other difficulties. Neither the time spent debugging nor

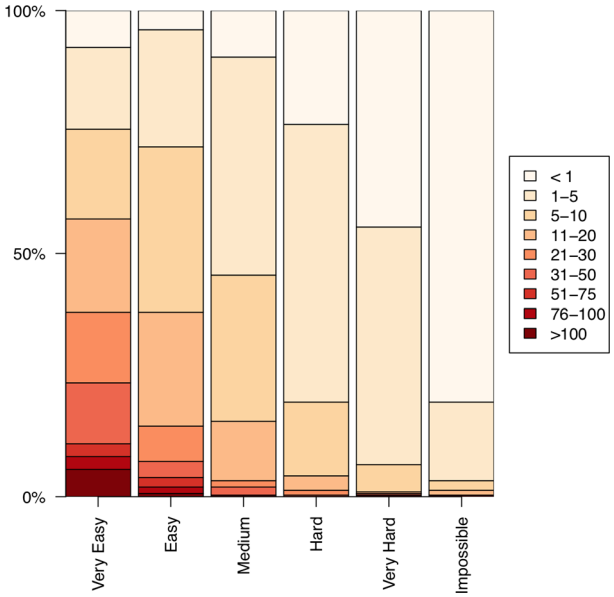

Fig. 9 How difficult are the bugs that you have to deal with (per month)?

the number of bugs encountered per month correlate with participant age, programming language used, debugging education, the debugging tools used, or the properties valued.

When asked how they feel about the change in difficulty of debugging software over the last 10 years, 133 said it got easier (44 %), 131 said it did not change (43 %), and 39 said it got harder (13 %). The predictions of the next 10 years show a similar distribution. One hundred and thirty-nine Participants said debugging will get easier (46 %), 117 said it will not change (39 %), and 47 said it will get harder (15 %). The predictions about the future correlate with the feeling about the past with a Pearson coefficient of 0.61.

4.3.5 Hardest bug

As already stated by Eisenstadt (1997), we can learn a lot from studying the most difficult bugs. These are the problems that profit most from improvements in debugging technology. For that reason, we asked our participants to position the hardest bug they ever faced in Eisenstadt's three dimensions. But first, we wanted to know how much time the participants needed to find and fix their hardest bug. Table 5 shows the results. Only a small portion of hardest bugs could be solved in one working day or less (17). The majority took 1 week or even longer (191) or was never fixed (46). *This shows that the most difficult bugs are serious issues taking up a lot of development time.*

1. Type of bug Table 6 shows the root cause participants could identify for their hardest bug.[2] The most frequent root cause was a design fault. *This confirms that there is a need for development processes that help finding design errors as early as possible.* The second most frequent root cause was parallel behavior, shortly followed by memory errors, and vendor responsibility. *This indicates that debugging parallel applications is especially*

[2] As early results of the survey have shown a large number of parallel behavioral bugs, we added it as a new choice that was not represented in Eisenstadt's original version.

Table 5 Time it took to fix the hardest bugs (number of answers given)

1 day	2 days	3–4 days	1 week	2 weeks	>2 weeks	never
17	21	38	70	31	80	46

Table 6 Root cause of the hardest bug (number of answers given)

Memory	Parallel	Vendor	Design	Init	Variable
42	53	41	82	9	3
Lexical	Ambiguous	User	Unknown	Other	
1	6	5	29	32	

Table 7 Most useful technique to find the hardest bug (number of answers given)

Stepping	Wrapper	Printf	Log diff	Breakpoints	Tool
54	5	33	12	38	15
Reading	Expert	Experiments	Not fixed	Other	
41	4	58	31	12	

difficult and may require specialized tools and methods not yet available. The high number of memory errors indicates that existing memory debuggers such as Valgrind or mtrace are either not powerful enough or not well known. Although one might suspect memory errors to be more prominent in C and C++, the distribution of used programming languages is not significantly different to the overall distribution. This may, however, be influenced by the fact that many participants named multiple languages. *The high number of bugs caused by vendor components indicates that external libraries also include a high number of failures.* Answers given as "other" are mostly bugs that are triggered by a combination of root causes acting together. It is noteworthy that many of the participants that said their hardest bug was never fixed could, however identify a root cause. We attribute this to bugs, that could be worked around or encapsulated so they did not impact the final program, but not really fixed. There is also the inverted case of bugs that were fixed according to the previous question but their root cause was unknown. These may be bugs that just vanished after a while, meaning they got fixed by accident.

2. Why was it especially hard to debug? Table 7 shows the technique that was most helpful to our participants, when they localized their hardest bug. The most helpful technique were controlled experiments. *This indicates that scientific debugging is not only easy to learn but also very useful to our participants.* The second most frequent answer was stepping through the program. This might be caused by the prominence of symbolic debuggers or by inspecting the program being useful to locate difficult bugs. *It might also indicate that some of these bugs may be caught faster if developers had other tools that could help them better.* The next most frequent answers were reading the code, setting breakpoints, and printf debugging. These indicate that hard bugs are those that require the developer to question his model of the program and rebuild or verify it. *Useful techniques mentioned as "other" include combinations of the provided categories, time to let the mind calm down, insights from non-experts, a redesign of program components, acquiring access to source code not available before, and creating a minimal failing example.*

Table 8 Main difficulty source for hardest bug (number of answers given)

Distance	Tools	Output	Assumption	Bad code	Unknown	Other
87	47	1	33	38	35	62

When relating the root cause of bugs with the most useful technique to find them, some patterns emerge. Applying a χ^2 test[3] indicates that these dimensions are dependent (χ^2 = 108, p value = 0.006):

- Specialized tools were useful to find memory errors and less useful for other types of bugs.
- Reading code was especially helpful to find bugs caused by parallel behavior.
- Controlled experiments were most useful to find bugs that were caused by vendor components.
- Almost all design errors have been fixed.

3. What technique turned out to be the most helpful to find it? Table 8 shows the reason why the hardest bugs were so difficult to find. The most frequent answer was the distance between the unexpected behavior and the root cause being high. *This indicates that tools that help uncovering the infection chain might especially support debugging efforts.* Such tools may for example be back-in-time debuggers and program slicing. Except for misleading program output, which was answered only once, the remaining categories share a similar number of answers. There was an very high number of "other" answers indicating that Eisenstadt?s categories are not sufficient: 14 participants blamed the inability to inspect or change relevant system parts; 14 identified their inability to reproduce the erroneous behavior in a controlled environment as the main difficulty; 10 participants named the necessity to change large parts of the system to verify their theory of the bug their highest obstacle; six participants blamed poor design, requirements, or documentation; six participants were hindered by a high complexity of the program; four candidates attributed the difficulty to a multitude of problems coming together; a long runtime of related test cases was mentioned twice; and one participant blamed company politics for keeping him from spending the necessary amount of time on the bug.

When relating the difficulty to the root cause, *the main difficulty when trying to find errors caused by parallel behavior or by vendor components seems to be the large distance between symptoms and root cause* (χ^2 = 101 p value = 0.036). Relating the difficulty to the most useful technique gave some more insights (χ^2 = 104 p value = 0.024). The interesting combinations were:

- Printf debugging was seldom useful to find the root cause with a large distance to symptoms.
- Stepping through the program or controlled experiments helped closing that gap.
- When the usual tools were hampered, most times only reading the code could help.
- When bad code limited the debugging success, stepping through the program provided the most help.

Changes in the last 15 years There are several differences to Eisenstadt's results. Bugs caused by a faulty design logic were more common in our survey, while memory errors

[3] A χ^2 test can be used for any kind of discretely categorized data with a large number of samples in each category. A large value indicates dependence and a low independence (Pearson 1900).

were less common. The new category of parallel errors was very common. While controlled experiments were very seldom among Eisenstadt's war stories, it was a very common answer in our survey. Eisenstadt's identified inapplicable tools as a main difficulty for hard bugs. While inapplicable tools were still very common in our survey, it was not much more common than other sources. Bad code was very seldom in Eisenstadt's analysis but the third most common shortly after hampered tools in our survey. This shows a significant shift in difficult bugs and the methods employed to find them, but a similar set of difficulties faced in the process.

4.3.6 Learning from past bugs

To find out what professional developers learn from their bugs, we asked them if they keep a log of their fixed bugs. One hundred and thirty-eight participants (46 %) said they add every bug to their log, 76 (25 %) said they add major bugs to the log, and 89 (29 %) said they do not keep one. Of those who use a log, 108 developers (50 %) said they add the solution to every bug to the log too. Ninety-six (45 %) said they add the solution to difficult bugs and 10 (5 %) said they do not add the solution to the log. *This may indicate a need among professional software developers to learn from the past and to avoid creating similar bugs again.*

To further investigate this, we asked our participants what they use these logs for. The results can be seen in Table 9. The most participants said they use it to perform code review. This makes sure the bug is removed and the cause is understood by at least one colleague. Using it to identify similar bugs in the future and to improve the development process are the next most frequent categories. Teaching colleagues and improving code quality mentioned our participants less often. Other answers included the coupling of logs with the version history of programs (10 times), a documentary function (5 times), using it to indicate a finished step in the development process (4 times), an extension of their own memory (3 times), the prevention of regressions (3 times), and communicating their work to customers. One participant explicitly mentioned not using it for anything.

Calculating the Pearson coefficients for the different uses of bug logs revealed some tendencies. People using their logs to improve code quality tend to also aim for improvement of their development process and vice versa (Pearson 0.32). Comparing new bugs to the log to identify similarities seems to be common among those participants who teach colleagues (Pearson 0.22) or try to improve their process (Pearson 0.24) using the bug log.

4.3.7 Bug prevention

We asked our participants to tell us how they check if their bugs are really removed. The results can be seen in Table 10. As might be expected, manual and automated testing are very common. Half of our participants also perform a code review, but code review was never the only method used. One hundred and fourteen participants (38 %) said they use three methods, 108 (37 %) use two of them, and 71 (23 %) use only one method. The

Table 9 For what reasons, do you keep logs? (number of answers given)

Teaching	Similar	Code review	Quality	Development process	Other
50	91	113	38	90	30

Table 10 How do you check the successful removal of bugs? (number of answers given)

Code review	Manual testing	Automatic testing	Other
166	275	208	7

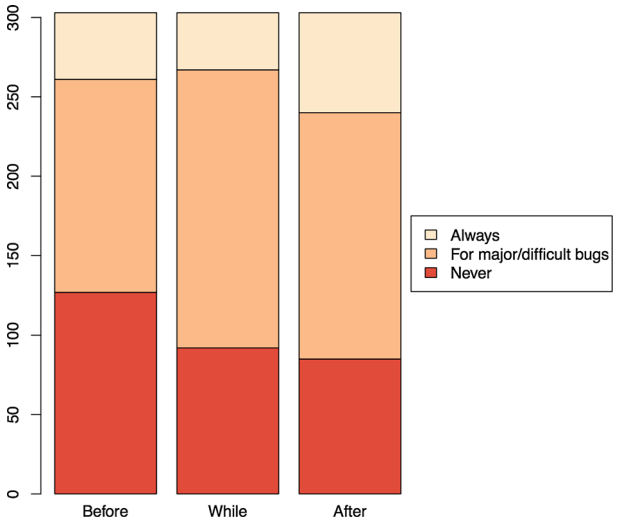

Fig. 10 Do you create tests during debugging?

remaining 10 participants (2 %) mentioned a quality assurance or testing department, benchmarks, user testing, and "letting time pass."

Figure 10 shows when developers create automated tests and if they do so at all. Test cases written after fixing the bug are slightly less common then test cases written upfront. Test cases written while debugging are in the middle. The minority of developers (45 participants, 15 %) never create any test cases. Most developers (171 participants, 56 %) create test cases for major or difficult bugs, but not for all. However, there is also a substantial number of developers (87 participants, 29 %) who always create test cases at least at one of these times. *Surprisingly, the creation of tests shows no correlations with the time spent on debugging nor with the number of bugs encountered.*

Figure 11 shows how often automated tests of different types are run. Functional tests are common among our participants, 175 (58 %) run them at least daily. Performance and destructive tests are less common, but still run regularly by 129 participants (43 %) each. There are also many participants (105 (35 %) and 91 (30 %)) who run these tests irregularly. Automated security tests are run less often. Only 72 (24 %) participants said they run them regularly, 129 (43 %) do not run them at all. *This might indicate a neglect for security, but can also attribute to the different needs of different software projects.* While functional specifications, a reasonable performance, and stability under unforeseen circumstances are a need for most software products, only those working with sensitive data, providing network operations, or potentially endangering people are concerned about security.

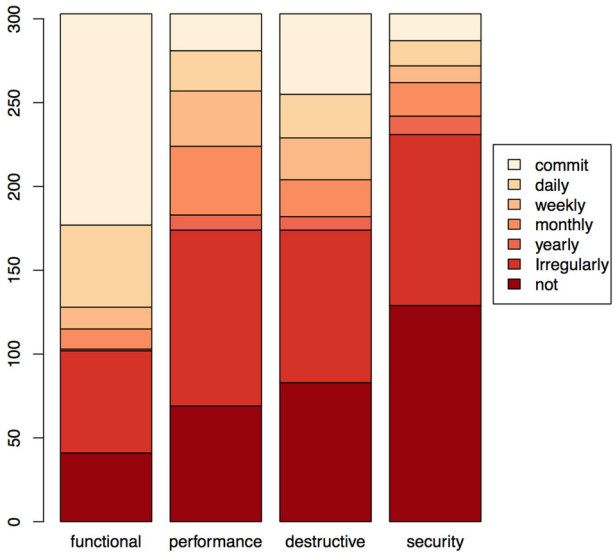

Fig. 11 How often do you run automated tests?

Figure 12 presents how often participants use automated tools to analyze different features of their product. The answers show the same tendency as in the previous question. *If participants adopted these methods at a regular part of their development process, they also decided to run them often.* While analyses are less common then automated tests, there is a substantial number of participants who analyze their source code for common pitfalls using a static code checker such as lint. We attribute this to the high efficiency of modern

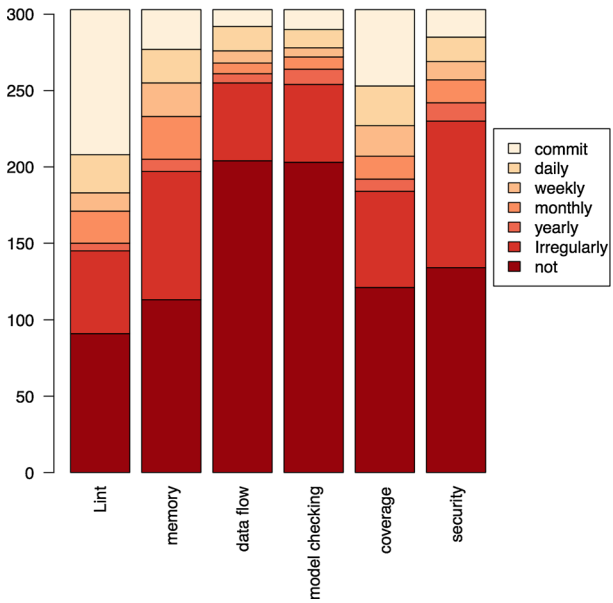

Fig. 12 How often do you perform automated analyses?

lint tools and their easy integration in IDEs or automated build processes. Test coverage analyses and checking for memory-related errors are almost as common, but not performed as regularly as lint. Security analyses are still performed by over half of the participants, but even less regular than coverage and memory analyses. Data flow analyses and model checking are not common among our participants. This might be attributed to the high run time of such tools and the development process overhead they can introduce. As it is the case for automated testing, the use of automated analysis shows no relation with the number of bugs encountered or the time spent debugging.

4.4 Research directions

The results of our online survey show that the field of debugging still has a lot of open issues. There are several things that can be further improved to easier the life of software developers. Here, we summarize our results and discuss valuable directions for future work.

Education The fact that participants having received debugging education did not answer significantly different may indicate that the existing education is not enough. While it is important to know how to use debugging tools, it is also important to know how to reason about a program and track down the source of a problem. There is a need to broaden this topic not only at universities. For example, at our institute, debugging is limited to one lecture as part of the "software engineering I" course. From our experience, the situation is often not much different at other locations. As debugging still requires a lot of time in software development, we argue to extend debugging education, to create practical exercises, and advanced tutorials. This will give students and software developers not only the right tools but also several systematic ways to tackle a problem.

Debugging tools In the last years, many researchers presented several new and very promising debugging tools. Unfortunately, these tools have not yet adopted by many professional software developers. On the other hand, popular debugging tools maturing very slowly with respect to new features and methods. For that reason, we need to mature our sophisticated tools for a large audience. In doing so, it is important to tackle the challenges of new debugging tools, namely, usability and overhead.

Workload Today as well as 15 years ago, debugging is a tedious task. We have not found a clear picture of improvement in debugging. It still costs developers a huge amount of their daily work time and bugs are as common and difficult to resolve as ever before. Thus, we conclude that there is still a lot to do. Research as well as practice need to incorporate their approaches, best practices, and experiences in order to reduce the workload for developers.

Hardest bug We have learned a lot from the hardest bug stories but the most important outcome for future work is the existence of a new root cause. Bugs being related to parallel behavior are the second most source. This points to a need for new tools and methods that better support debugging of parallel applications. The many different patterns for creating parallel programs and the difficulties in observing multiple control flows at the same time hinder current debugging tools to support developers effectively. Further investigation of this special area of program debugging could provide valuable insights.

Learning from past bugs Our study revealed that bug logs are an important tool. This source of information allows developers to share their experiences and to prevent similar bugs in advance. However, maintaining these logs requires additional time and carefulness. For that reason, they are mostly done, if at all, only for the most difficult bugs. We think that a better support in logging can help developers in training colleagues for localizing

similar bugs more easily. For example, the idea of an automated logging mechanism seems to be promising as it records all debugging steps without much effort. After that, developers can share their results with the team for teaching purposes.

Bug prevention We could not prove any relation between automated testing and analyses and the number of bugs encountered or the time spent on debugging. While this could indicate ineffectiveness of the established automated methods with respect to debugging, we rather argue that testing and analyses enhance the development process and its outcome in general. With the help of such tools, developers not only produce more and complex code but also they help them to reveal more bugs in their more difficult code. It might be interesting to study the effects of using automated bug prevention methods with respect to code quality and developer effectiveness in more detail.

4.5 Threats to validity

The main threat to the validity of our results is the limited number of participants. While 303 participants can provide interesting insights, a larger number is required to make reliable statements. Many of the Pearson coefficients we calculated are small, meaning they cannot be relied on to draw a final conclusion on the relation of the variables. However, they are distinct from zero and so indicate at least a positive correlation.

Additionally, all answers depend on self-reflectance of the participants, which may vary in precision and correctness. While this did probably not create a systematic error, it may have increased the noise to levels that make correlation checks incorrect. The most prominent tasks of this type are rating the importance of tool properties and estimating the change in difficulty of debugging software.

It may be worth noting that it is disputable if participants under the age of 20 can be considered professional software developers. However, at the beginning of the study we explicitly asked only professional software developers to continue, so these participants probably consider themselves such. Although they are probably inexperienced, they may work as interns or contribute to open-source projects.

There is some information, that we did not acquire, although it would be useful. For example, we did not collect the kind of software development our participants are working in. This disables us from analyzing the difference between web developers, backend developers, mobile application developers, and probably more groups.

Although we tested the questionnaire internally before starting the online survey, there remain some flaws in the design of the existing questions:

- When asking the participants, what programming languages they use, a lot of people gave too many answers. This made it very difficult to analyze the programming language used for correlations. A limit on the number of languages named could help to reduce this problem.
- In the question about the knowledge of debugging tools there was the option “I tried.” Some participants told us, they felt this option was poorly chosen, because it suggested rejecting the tool. “I use sometimes” might have been a better choice.
- When analyzing the importance score participants assigned to different properties of debugging tools, we were not able to analyze if the order of importance shows patterns. There are more properties then degrees of importance and many developers chose the same degree for multiple answers. Asking them to order the properties instead of assigning an importance score may have been more useful.

- Similarly, participants disagreed what counts as a bug log, varying from sophisticated bug tracking software to commit messages.
- Some of the answers given as "other" in all three of Eisenstadt's dimensions included answers that fit in one of the provided categories. This indicates that the categories were not explained clearly enough in the survey, causing confusion in the participants. Nevertheless, we ordered these answers manually in the end.

Another indicator for flaws in the survey design is the number of people who did not complete the survey. Out of 713 participants who started the survey, only 303 completed it. One hundred and ninety-nine potential participants did not answer any question. Another 34 did not want to answer questions about their companies size. Ninety-four stopped before answering the questions on tool knowledge and tool properties. Thirty-five would not tell about their workload. Thirty-four did not answer the questions concerning the hardest bug. Twelve did not answer the bug prevention question. The remaining four would not answer the questions on debugging related education. Almost all questions were mandatory, some participants told us that they did not like that. While we used this feature to prevent holes in the data, it might have prevented us from reaching a larger number of participants. It is also possible that some participants chose random answers because they were forced to answer a question they did not want to. This might contribute to some noise.

5 Conclusion

In this paper, we presented our results of studying debugging behavior 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 localization techniques. Based on these results, we created an online survey to consolidate and expand our results. A total of 303 software developers took part in this survey and answered all the questions.

Only half of our participants mentioned receiving debugging education, indicating that educators still assume that debugging is a minor part of software development or that students will learn it by themselves. However, there also was no significant difference between participants with and without education, indicating that the existing courses and trainings are indeed not more effective in teaching important debugging skills than self-learning.

By placing the hardest bugs in Eisenstadt's dimensions of bug classification, we were able to show that the most difficult bugs are related to erroneous program design or parallel behavior. Most participants named a large distance between root cause and observable failure as the main difficulty of finding bugs. This indicates that developers are in need of tools that support them in analyzing the interactions of different parts of the program, even if those run in parallel. We could also observe that different tools and methods were useful for different types of bugs, judging which strategy and tool to apply may be one important skill to learn when trying to reduce debugging time. Our results also vary from Eisenstadt's in many ways: we found fewer memory errors and the new category of parallel errors; we

revealed that it is now very common to apply controlled experiments, and, finally, the main difficult sources for bugs are not only inapplicable tools anymore but causes such as bad code have also strong influence. This shows that debugging is an ever changing field, where it is necessary to reevaluate the challenges and opportunities from time to time.

References

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

Agrawal, H., Horgan, J., London, S., & Wong, W. (1995). Fault localization using execution slices and dataflow tests. In *conference on software reliability engineering* (pp. 143–151).

Ahmadzadeh, M., Elliman, D., & Higgins, C. (2005). An analysis of patterns of debugging among novice computer science students. *ACM SIGCSE Bulletin*, *37*(3), 84–88.

Artzi, S., Dolby, J., Tip, F., & Pistoia, M. (2010). Practical fault localization for dynamic web applications. In *international conference on software engineering* (pp. 265–274). ACM.

Arumuga Nainar, P., & Liblit, B. (2010). Adaptive bug isolation. In *international conference on software engineering* (pp. 255–264). ACM.

Baah, G. K., Podgurski, A., & Harrold, M. J. (2010). Causal inference for statistical fault localization. In *international symposium on software testing and analysis* (pp. 73–84). ACM.

Ballou, M. C. (2008). Improving software quality to drive business agility. IDC Survey and White Paper.

Chmiel, R., & Loui, M. C. (2004). Debugging: From novice to expert. *ACM SIGCSE Bulletin*, *36*(1), 17–21.

Cleve, H., & Zeller, A. (2005). Locating causes of program failures. In *international conference on software engineering* (pp. 342–351). ACM.

Dallmeier, V., Lindig, C., & Zeller, A. (2005). Lightweight defect localization for java. In *European conference on object-oriented programming* (pp. 528–550). Springer.

Eisenstadt, M. (1997). My hairiest bug war stories. *Communications of the ACM*, *40*(4), 30–37.

Gould, J. D. (1975). Some psychological evidence on how people debug computer programs. *International Journal of Man-Machine Studies*, *7*(2), 151–182.

Gould, J. D., & Drongowski, P. (1974). An exploratory study of computer program debugging. *The Journal of the Human Factors and Ergonomics Society*, *16*(3), 258–277.

Grötker, T., Holtmann, U., Keding, H., & Wloka, M. (2012). *The developer's guide to debugging* (2nd ed.). NewYork: Self-publishing company.

Gupta, N., He, H., Zhang, X., & Gupta, R. (2005). Locating faulty code using failure-inducing chops. In *international conference on automated software engineering* (pp. 263–272). ACM.

Hailpern, B., & Santhanam, P. (2002). Software debugging, testing, and verification. *IBM Systems Journal*, *41*(1), 4–12.

Hanks, B., & Brandt, M. (2009). Successful and unsuccessful problem solving approaches of novice programmers. *ACM SIGCSE Bulletin*, *41*(1), 24–28.

James, S., Bidgoli, M., & Hansen, J. (2008). Why sally and joey can't debug: Next generation tools and the perils they pose. *Journal of Computing Sciences in Colleges*, *24*(1), 27–35.

Janssen, T., Abreu, R., & van Gemund, A. J. (2009). Zoltar: A toolset for automatic fault localization. In *international conference on automated software engineering* (pp. 662–664). IEEE Computer Society.

Jeffrey, D., Gupta, N., & Gupta, R. (2008). Fault localization using value replacement. In *international symposium on software testing and analysis* (pp. 167–178). ACM.

Jiang, L., & Su, Z. (2007). Context-aware statistical debugging: From bug predictors to faulty control flow paths. In *international conference on automated software engineering* (pp. 184–193). ACM.

Jones, J. A., Bowring, J. F., & Harrold, M. J. (2007). Debugging in parallel. In *international symposium on software testing and analysis* (pp. 16–26). ACM.

Kernighan, B. W., & Plauger, P. J. (1978). *The elements of programming style* (Vol. 1). NewYork: McGraw-Hill.

LaToza, T. D., & Myers, B. A. (2010). Developers ask reachability questions. In *international conference on software engineering*, vol. 1, (pp. 185–194). IEEE.

Lencevicius, R. (2000). On-the-fly query-based debugging with examples. arXiv.

Lewis, B. (2003). Debugging backwards in time. In *proceedings of the international workshop on automated debugging, AADEBUG* (pp. 225–235). Arxiv.

Liblit, B., Naik, M., Zheng, A. X., Aiken, A., & Jordan, M. I. (2005). Scalable statistical bug isolation. *ACM SIGPLAN Notices*, *40*(6), 15–26.

Lieberman, H. (1997). The debugging scandal and what to do about it (introduction to the special section). *Community ACM*, *40*(4), 26–29.
Liu, C., Yan, X., Fei, L., Han, J., & Midkiff, S. P. (2005). Sober: Statistical model-based bug localization. *ACM SIGSOFT Software Engineering Notes*, *30*(5), 286–295.
Metzger, R. C. (2004). *Debugging by thinking: A multidisciplinary approach*. NewYork: Elsevier Digital Press.
Murphy, L., Lewandowski, G., McCauley, R., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: The good, the bad, and the quirky-a qualitative analysis of novices' strategies. *ACM SIGCSE Bulletin*, *40*(1), 163–167.
Park, S., Vuduc, R. W., & Harrold, M. J. (2010). Falcon: Fault localization in concurrent programs. In *international conference on software engineering* (pp. 245–254). ACM.
Pearson, K. (1895). Notes on regression and inheritance in the case of two parents. In *proceedings of the royal society of London* (pp. 240–242). The royal society.
Pearson, K. (1900). On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. In *philosophical magazine* (pp. 157–175).
Perscheid, M. (2013). Test-driven fault navigation for debugging reproducible failures. Ph.D. thesis, Hasso Plattner Institute, University of Potsdam.
Renieres, M., & Reiss, S. P. (2003). Fault localization with nearest neighbor queries. In *international conference on automated software engineering* (pp. 30–39). IEEE.
Vessey, I. (1985). Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, *23*(5), 459–494.
Weiser, M. (1982). Programmers use slices when debugging. *Communications of the ACM*, *25*(7), 446–452.
Wong, W. E., & Debroy, V. (2009). A survey of software fault localization. Department of Computer Science, University of Texas at Dallas, Technical Report UTDCS-45-09.
Yilmaz, C., Paradkar, A., & Williams, C. (2008). Time will tell: Fault localization using time spectra. In *international conference on software engineering* (pp. 81–90). ACM.
Zeller, A. (2002). Isolating cause-effect chains from computer programs. In *symposium on foundations of software engineering* (pp. 1–10). ACM.
Zeller, A. (2009). *Why programs fail: A guide to systematic debugging*. Burlington: Morgan Kaufmann.
Zhang, X., Gupta, N., & Gupta, R. (2006). Locating faults through automated predicate switching. In *international conference on software engineering* (pp. 272–281). ACM.

Michael Perscheid is a researcher with the SAP Innovation Center Potsdam. Before joining SAP in 2014, he was a postdoctoral researcher and Ph.D. student in computer science with the Software Architecture Group, led by Prof. Dr. Robert Hirschfeld, at the Hasso-Plattner-Institute (University of Potsdam).

Benjamin Siegmund is a software developer with Demandware. He received a master's degree from the Hasso-Plattner-Institute, University of Potsdam, Germany.

Marcel Taeumel is a research assistant in the Software Architecture Group at the Hasso-Plattner-Institute (HPI) and a member of the HPI Research School on Service-Oriented Systems Engineering. He received a master's degree from the Hasso-Plattner-Institute, University of Potsdam, Germany.

Robert Hirschfeld is a Professor of Computer Science at the Hasso-Plattner-Institut (HPI) at the University of Potsdam. He received a Ph.D. in Computer Science from the Technical University of Ilmenau, Germany. See also http://www.hpi.uni-potsdam.de/swa/.