How developers debug

Debugging software is an inevitable chore, often difficult and more time-consuming than expected, giving it the nickname the “ dirty little secret of computer science.” Surprisingly, we have little knowledge on how software engineers debug software problems in the real world, whether they use dedicated debugging tools, and how knowledgeable they are about debugging. This study aims to shed light on these aspects by following a mixed-methods research approach. We conduct an online survey capturing how 176 developers reflect on debugging. We augment this subjective survey data with objective observations from how 458 developers use the debugger included in their Integrated Development Environments (IDEs) by instrumenting the popular ECLIPSE and INTELLIJ IDEs with our purpose-built plugin WATCHDOG 2.0. To better explain the insights and controversies obtained from the previous steps, we followed up by conducting interviews with debugging experts and regular debugging users. Our results indicate that the the IDE-provided debugger is not used as often as expected, since “printf debugging” remains a feasible choice for many programmers. Furthermore, both knowledge and use of advanced debugging features are low. Our results call to strengthen hands-on debugging experience in Computer Science curricula and can and have already influenced the design of modern IDE debuggers.

How Developers Debug

Moritz Beller[§]
Delft University of Technology,
The Netherlands
m.m.beller@tudelft.nl

Niels Spruit[§]
Delft University of Technology,
The Netherlands
spruit.niels@gmail.com

Andy Zaidman
Delft University of Technology,
The Netherlands
a.e.zaidman@tudelft.nl

***Abstract*—Debugging software is an inevitable chore, often difficult and more time-consuming than expected, giving it the nickname the "dirty little secret of computer science." Surprisingly, we have little knowledge on how software engineers debug software problems in the real world, whether they use dedicated debugging tools, and how knowledgeable they are about debugging. This study aims to shed light on these aspects by following a mixed-methods research approach. We conduct an online survey capturing how 176 developers reflect on debugging. We augment this subjective survey data with objective observations from how 458 developers use the debugger included in their Integrated Development Environments (IDEs) by instrumenting the popular ECLIPSE and INTELLIJ IDEs with our purpose-built plugin WATCHDOG 2.0. To better explain the insights and controversies obtained from the previous steps, we followed up by conducting interviews with debugging experts and regular debugging users. Our results indicate that the the IDE-provided debugger is not used as often as expected, since "printf debugging" remains a feasible choice for many programmers. Furthermore, both knowledge and use of advanced debugging features are low. Our results call to strengthen hands-on debugging experience in Computer Science curricula and can and have already influenced the design of modern IDE debuggers.**

***Index Terms*—Debugging, Testing, Eclipse, IntelliJ, WatchDog**

I. INTRODUCTION

Debugging, the activity of identifying and fixing faults in software [1], is a tedious but inevitable chore in almost every software development project [2]. Not only is it inevitable, but Zeller also states that it is difficult and, therefore, consumes much time, often more than creating the bogus piece of software in the first place [3]. During debugging, software engineers need to relate an observed failure to its underlying defect [4]. To complete this step efficiently, they need to have a deep understanding and build a mental model of the software system at hand. This is where modern debuggers come in: they can aid software engineers in gathering information about the system, but they still require them to select relevant information and perform the reasoning.

While scientific literature is rich in terms of proposals for new (automated) debugging approaches, e.g., [3], [5]–[8], there is a gap in knowledge of how practitioners actually debug. Debugging has thus remained *the dirty little secret of computer science* [9]. How and how much do software engineers debug at all? Do they use modern debuggers? Are they familiar with their capabilities of IDE debuggers? And which other tools and strategies of the trade do they know?

[§]Contributed equally and are both first authors.

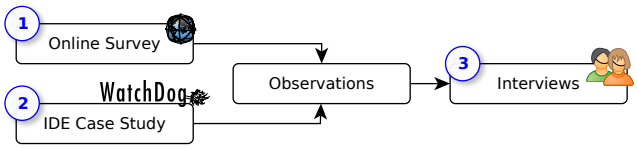

Fig. 1. Overview of our general research design.

The lack of knowledge on developers' debugging behavior is in part due to an all too human characteristic: admitting, demonstrating, and letting others do research on how one approaches what are essentially one's own faults is a difficult situation for both developer and researcher. But, however unpleasant, by continuing to keep debugging practices secret, we miss an important opportunity for improvement of software engineering and potentially a large cost driver of software: failures, which developers were unable to prevent, cost the US economy $59.5 billion a year according to a 2002 study [10].

Knowledge on how developers debug can help researchers to invent more relevant prototypes and techniques, educators to adjust their teaching on debugging, and tool creators to tailor debuggers to the actual needs of developers. To help make the art of debugging an open secret, we performed a large-scale behavioral field study, specifically in the area of what developers think about debugging and how they debug in their Integrated Development Environment (IDEs). The following questions steer our research:

RQ1 How do developers reflect on debugging?

RQ2 When and how long do developers debug?
RQ2.1 How much of their active IDE time do developers spend on debugging (compared to other activities)?
RQ2.2 What is the (average) frequency and length of the debugging sessions performed by developers?
RQ2.3 At what times do developers launch the IDE debugger?
RQ2.4 Do long files require more debugging?
RQ2.5 How is debugging effort distributed across classes?
RQ2.6 Do developers who test more have to debug less?
RQ2.7 Do experienced developers have to debug less?
RQ2.8 Do developers often step over the point of interest?

RQ3 How do individual debugger users and experts interpret our findings from *RQ1* and *RQ2*?

Research Design

In order to answer these research questions, we employ a multi-faceted research approach outlined in Figure 1. We conducted an online survey to capture developers' opinions on debugging and obtain an overview of the state of the practice ①. Simultaneously, we began using our WATCHDOG 2.0 infrastructure to track developers' fine-grained debugging activities in the IDE ②. After we compared both case studies, we came up with a list of several, potentially conflicting observations that needed further explanation. To help us explain the findings in depth, we conducted interviews with developers, some of whom are actively developing debugging tools ③.

II. BACKGROUND

This section describes related work and research tools.

A. Related Work

In this section, we give an overview of debugging tools and show how existing research relates to this paper.

Debugging Tools. When referring to "debuggers," most programmers usually mean *symbolic debuggers* like GDB [11]. Such debuggers allow them to specify points in the program where the execution should be suspended, called *breakpoints*. A typical symbolic debugger supports different *types of breakpoints*, such as *line, method*, or more advanced *exception* or *class prepare breakpoints*, and options to further refine the moment when the program should be suspended [12]. Examples include specifying a condition *(conditional breakpoint)* or a *hit count* for program execution to suspend. A *suspension policy* defines whether the entire program or just one thread should be suspended once a breakpoint is hit.

Once the program is suspended, developers can use symbolic debuggers to watch or inspect variables, work through the call stack, line-wise step through the code, or evaluate arbitrary expressions [11], [12]. Originating from command line symbolic debuggers like GDB, several graphical symbolic debuggers appeared, e.g. DDD [13]. Most symbolic debugging features have today been integrated in the debuggers shipped as part of *Integrated Development Environments* (IDEs) like ECLIPSE and INTELLIJ. This study focuses on such IDE debuggers and how developers use them.

Debugging Process. Researchers have developed systematic process descriptions of debugging and come up with recommendations to reduce the time programmers have to spend on finding and fixing a defect that causes a program failure. We want to also find out whether developers explicitly or implicitly use such debugging strategies inspired by the scientific method; for example, Zeller's *TRAFFIC* approach [3] comprises seven steps that cover every action in the debugging process, from the discovery of a problem until the correction of the defect. Three of those steps regard "the by far most time consuming" Find-Focus-Isolate loop, as developers often need to follow them iteratively to find the root cause of a failure. Therefore, much research has gone into techniques to, at least partially, automate this loop to reduce debugging effort.

Debugging Techniques. Arguably the most researched debugging technique is *delta debugging*, which can be used to systematically narrow down possible failure causes by comparing a successful and an erroneous program execution [14]. Other types of debugging technique include *slicing* [3], focusing on *anomalies* [3], *mining dynamic call graphs* [15], *statistical debugging* [16], *spectra-based fault localization* [5], *angelic debugging* [6], *data structure repair* [8], *relative debugging* [17], *automatic breakpoint generation* [18] and *automatic program fixing using contracts* [7]. Finally, several combinations of the techniques exist [19]–[23]. Orso presents a detailed explanation and evaluation of some of these automated debugging techniques [24]. However, because automated debugging techniques have apparently not yet reached the mainstream debugging practices of developers and are not part of the IDE debuggers in our study, we do not discuss them further.

Empirical Debugging Evidence. To the best of our knowledge, only a handful of research efforts exist that empirically evaluated how developers debug. Siegmund et al. studied the debugging practices of professional software developers [4], the research effort that is perhaps closest to ours, but different in study population and methodology. They followed eight software developers across four different companies through think-aloud protocols and short interviews. Their results indicate that none of the developers had any debugging-specific education or training. Furthermore, "all developers use[d] a simplified scientific method," which consists of formulating and verifying hypotheses as described in TRAFFIC [3]. "All participants [were] proficient in using symbolic debuggers" and preferred them over print statements. None of the developers were aware of back-in-time debuggers [25], which allow to step back in the execution history and not only forward, the automated techniques described above, or more advanced symbolic debugging features, like conditional breakpoints.

Subsequently, Siegmund et al. created an online survey on "debugging tools, workload, approach and education" [26]. Based on the answers of 303 respondents, they found that "debugging education is still uncommon, but more [...] courses started including it recently." Furthermore, most participants only use "older debugging tools such as printf, assertions and symbolic debuggers." Concurrency issues and external libraries often seem to be the root causes of the hardest bugs.

Piorkowski et al. studied qualitatively how programmers forage for information [27], [28]. They found that developers spent half of their debugging time foraging for information. This complements our study as it also shows what parts of the IDE are often used for finding information during debugging which might impact the debugging features developers use.

B. Related Tools

Swarm Debugging. Petrillo et al. developed the *Swarm Debug Infrastructure* (SDI), which "provides [Eclipse] tools for collecting, sharing, and retrieving debugging data" [29]. Developers can utilize the collective swarm intelligence of previous debug sessions to "navigate sequences of invocation methods" and "find suitable breakpoints." Petrillo et al. evaluated SDI

in a controlled experiment, in which 10 developers were asked to debug into three faults of a program. Developers toggled only one or two breakpoints per fault and there were no correlations between numbers of breakpoints and developers' expertise and task complexity. Instead, developers followed diverging debugging patterns. Our approach to *RQ2* is technically similar to SDI in that we instrument ECLIPSE. To broaden generalizability, we also support a second IDE, INTELLIJ, and performed a longitudinal field study of how dozens of developers debug in the wild.

III. DEBUGGING SURVEY

In this section, we describe our online survey.

A. Research Methods

Survey Design. To investigate developers' self-assessed knowledge on debugging for *RQ1*, we set up a survey.[1] It consists of 13 short questions (*SQ1-SQ13*) organized in four sections; the first gathers general information about the respondents, such as programming experience and favorite IDE. The second asks if and how respondents use the IDE-provided debugging infrastructure. Developers who do not use it were asked for the reason why, while others got questions on specific debugging features, thus assessing how well the respondent knows and uses several types of breakpoints. In addition, we asked questions about other debugging features ranging from stepping through code to more advanced features like editing at run time (hot swapping). The third part, presented to all respondents, assessed the importance of codified tests in the debugging process; we gauged whether the participant uses tests for reproducing bugs, checking progress, or to verify possible bug fixes. *SQ13* was an open, non-mandatory question on participants' opinion to the statement "the best invention in debugging still was printf debugging." We included it because research on survey design has shown that posing a concrete, controversial statement that evokes strong opinions leads to more insightful answers [30]. Before publicly releasing the survey, we made several iterations and ran it across six externals.

Card Sort. To gain an overview of the topics that concern developers, we performed an *open card sort* [31] on *SQ13*. The first two authors individually built and then mutually agreed on a set of tags from a sub-sample of responses. After labeling all responses (possibly with multiple labels), the third author sampled 20% of responses, tagged them independently and we converged our tag sets to arrive at a homogeneous classification of all answers.

Dependency Analysis. To gain insights into the correlation between survey answers, we performed statistical tests. For *SQ7-12*, we had to convert each categorical answer to an ordinal scale using a linear integer transformation on its rank. This was sound because our predefined answer options have a naturally ranked order ("I don't know" = 1, "I know" = 2, ...). We then computed a pair-wise *Pearson Chi-Squared (χ^2) test of independence* [32], as we are dealing with categorical variables. If variables depended on each other ($\alpha = 0.05$), we calculated the strength of their relationship with a *Spearman rank-order correlation test* for non-parametric distributions [33]. For interpreting the results of dependency analyses ρ, we use Hopkins' guidelines [34]. They assume no correlation for $0 \leq |\rho| < 0.3$, a weak correlation for $0.3 \leq |\rho| < 0.5$, a moderate correlation for $0.5 \leq |\rho| < 0.7$ and a strong correlation for $0.7 \leq |\rho| \leq 1$.

Recruitment of Subjects. To attract respondents (*SR*), we spread the link to the survey through social media, especially Twitter, and via an in-IDE WATCHDOG registration dialog, advertising a raffle with three 15 Euro Amazon vouchers.

Study Subjects. We attracted 176 software developers who completed our survey. The majority of them have at least three years of experience in software development, with a third over 10 years (< 1 year: 2.8%, 1-2 years: 6.8%, 3-6: 31.8%, 7-10: 21.6%, > 10 years 36.9%). 84.1% indicated that they use Java, followed by 55.1% for JavaScript and 39.2% for Python. The languages PHP, C, C++ and C# were each selected by around 25% of participants, followed by R (16.5%), Swift (6.3%) and Objective-C (5.1%). Finally, 44 developers indicated the use of another language (24 different in total), of which Scala (11) and Ruby (8) prevail. The most used IDEs are Eclipse (31.8%), IntelliJ (30.7%), and Visual Studio (11.9%).

B. Results

Analysis of Survey Answers. In our first question, 143 developers (81.3%) indicated that they use the IDE-provided debugging infrastructure, 15 (8.5%) indicated that they do not, and 18 developers (10.2%) that their selected IDE does not have a debugger. Besides using the IDE debugger, respondents indicated they examine log files (72.2%), followed closely by using print statements (71.6%). Other answers included the use of an external program (21.0%), or additional other techniques (30.1%). 19 developers indicated the use of a complementary method, of which adding or running tests and using web development tools built into the browser were mentioned most (both four times).

> *SO1:* Most developers use the IDE-provided debugging infrastructure in conjunction with log files and print statements.

Of the 15 developers not using the debugging infrastructure, 8 think that print statements and 6 that techniques other than print statements are more effective or efficient, 6 use an external program they find more effective or efficient, while 4 do not know how to use the debugger.

> *SO2:* Developers not using the IDE-provided debugging infrastructure find external programs, tests, print statements, or other techniques more effective or efficient.

The 143 developers using an IDE debugger were asked more detailed questions on whether they know and use specific debugging features. The Likert scale plots in Figure 2 show that most developers are familiar with line, exception, method and field breakpoints, while temporary line breakpoints and class prepare breakpoints are known by fewer developers.

[1] https://goo.gl/AFLojW

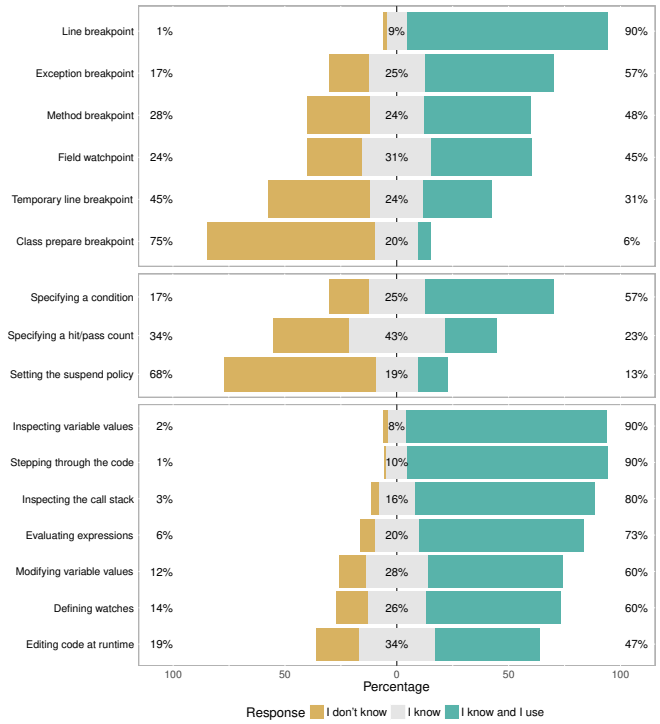

Fig. 2. Answer distribution in *SQ7-9* on breakpoint types, breakpoint options, and debugging features ($n = 143$).

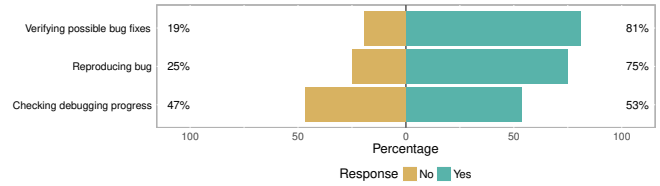

Fig. 3. Answer distribution in *SQ10-12* on unit tests ($n = 176$).

The vast majority of developers also uses line breakpoints, but other breakpoint types are used by less than half of the respondents; Class prepare breakpoints are used by almost none.

SO3: Line breakpoints are used by the vast majority of developers. More advanced types are unknown to most.

Figure 2 indicates that the majority of developers specify conditions on breakpoints. However, specifying the hit count or setting a suspend policy are both known and used less.

SO4: Most developers answered to be familiar with breakpoint conditions, but not with hit counts and suspend policies.

The results in Figure 2 show that over 80% of the developers seem to know all major debugging features found in modern IDEs, strengthening Siegmund's findings [4]. The more advanced features, like defining watches or a suspend policy, seem to be known and used less.

Figure 3 visualizes the use of codified tests throughout the debugging process based on all 176 responses. It indicates that tests are often used at the start and end of the debugging process, for reproducing bugs and verifying bug fixes, respectively, and slightly less throughout the debugging process.

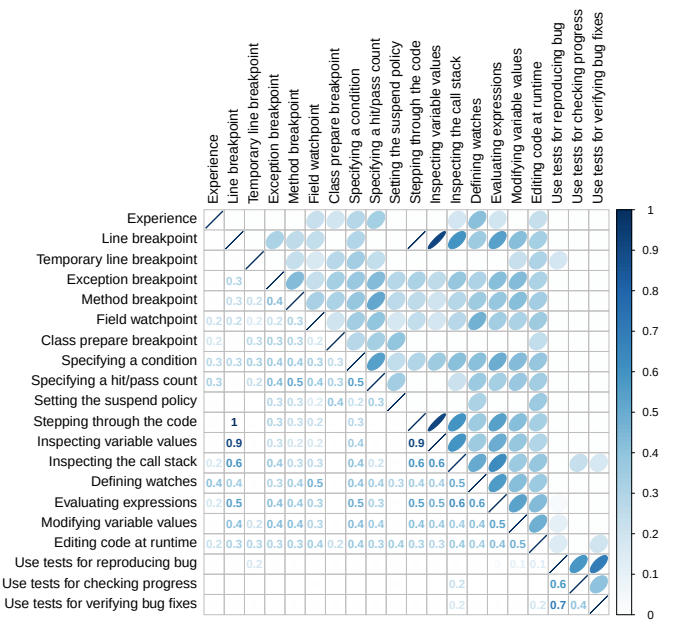

Fig. 4. Correlation analysis of *SQ1, SQ7-12* ($n = 143$).

SO5: Survey answers indicate testing is an integral part of the debugging process, especially at the beginning and end.

Dependency Analysis. Examining our survey answers for dependencies allows us to understand how certain answer relate, for example whether and how strongly programmer experience correlates with the use of debugger features like breakpoints, watches or the use of testing to guide debugging.

Figure 4 visualizes the results of both computation on the $n = 143$ survey responses that indicated to use the debugger. The more intense the color and the flatter the ellipses, the higher is the strength of the correlation, also reported numerically below the diagonal. Empty cells correspond to non-significant results of the χ^2 test. We applied no Bonferroni correction as its use is highly controversial [35], especially for overview purposes.

Based on the results in Figure 4, we find that there is no correlation between the use of an IDE debugger or (unit) tests for debugging and experience in software development. There is a weak correlation between experience and specifying hit counts and a moderate correlation between experience and the usage of watches during debugging.

SO6: Experience has limited to no impact on the usage of the IDE-provided debugging infrastructure and tests.

We also find that there is a moderate correlation between the use of tests at the beginning and end of the debugging process to reproduce and verify bug fixes, and a weak to moderate correlation between using tests at the beginning or end and throughout the process for checking progress.

SO7: Developers who use tests for reproducing bugs are likely to use them for checking progress and very likely to use them for verifying bug fixes as well.

Figure 4 shows many other correlations between the knowledge and use of several debugging features. Most notably,

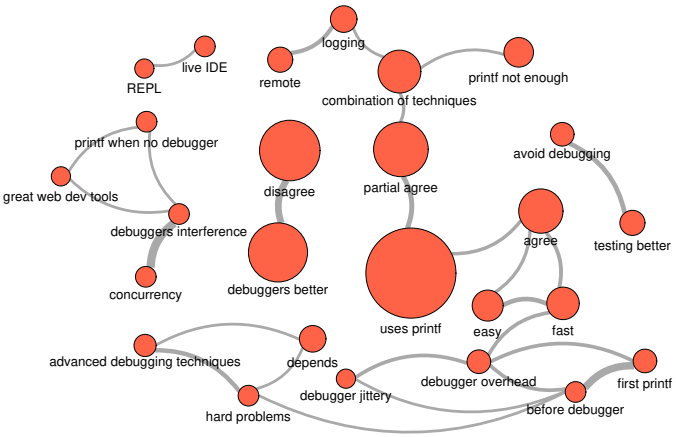

Fig. 5. Intuitive representation of the tag network extracted from *SQ13*.

there is a quite strong correlation between the following three features: line breakpoint, stepping through code and inspecting variable values.

SO8: Developers that use line breakpoints also step through the code and inspect variable values during debugging.

Card Sorting. In total, 108 respondents gave a response to the statement that "the best invention in debugging still was printf debugging." In the open card sorting process, we identified 34 different tags. To understand important topics and their co-occurrence, we use an intuitive graph-based representation of the tag structure. Vertices correspond to the tags and undirected, weighted edges to the strength of relation between two tags. The size of the vertices in is determined by the occurrence frequency of the tag, while the weight of the edges is determined by the relative number of co-occurrences. To ease the interpreting the graph, (1) we normalized the weights of the edges based on the occurrence frequencies of its end points, (2) we filtered out all edges with a low normalized weight and (3) we removed vertices that did not have any outgoing or incoming edge. The resulting graph in Figure 5 allows us an intuitive understanding and overview of responses and how they relate to each other, without having to read hundreds of responses [36].[2] The tags 'debugger jittery', 'debugger overhead' and 'debugger interference' mean that respondents think debuggers have too much impact on the thinking process, performance or program execution, respectively. 'First printf' means that developers first use printf debugging and 'before debugger' indicates that developers use some other technique(s) before using the debugger.

As Figure 5 indicates, there was a strong dichotomy between survey respondents: Some enthusiastically agreed with our statement ("Totally agree!", *SR13, SR23*), while others rejected it, stating that "[p]eople saying that never learned how to use a debugger" (*SR54*). Many respondents were also in-between, seeing benefits and drawbacks, like *SR10*: "Print is often most flexible but often least efficient." Developers indicated to use printf debugging as an ad-hoc, universal technique that is easy to do and often the first step in a possibly longer debugging strategy. However, sometimes it is not enough, as a combination of techniques is required. Developers mostly seemed to agree that IDE debuggers are methodologically superior to print statements. However, their answers also pointed to problems with IDE debuggers: they are sometimes too jittery, provide too many features and are not suited for concurrent debugging as they interfere too much. Moreover, their complicated UI can get in the way of working (fast). Instead of pintf debugging, developers seemed to prefer a live IDE with a console that has a read-eval-loop (REPL). Summarizing this discussion nicely, *SR75* concluded that "printf is travelling by foot, a GUI debugger is travelling [by] plane. You can go to more places by foot, but you can only go that far."

[2]We explicitly avoided statistical tests. Given open-ended survey answers as the source, the meaning of such tests is unclear at best, or can convey a false sense of statistical preciseness and completeness at worst.

IV. IDE Field Study

In this section, we describe our WatchDog field study.

A. Study Methods

Data Collection. To investigate the debugging habits of developers in the IDE, we extended our existing WatchDog [37]–[39] infrastructure to also track developers' debugging behavior for *RQ2*, resulting in WatchDog 2.0 for both Eclipse and IntelliJ. The original WatchDog was used to verify common expectations and beliefs about testing [37], [38]. WatchDog is technically centered around the concept of *intervals* that capture the start and end of common development activities like reading and writing code as well as running JUnit tests. We extended its interval concept to cover debugging sessions and introduce singular debugging events like breakpoint additions, changes and removals.

Analysis Methods. To analyze the data collected with WatchDog 2.0, we integrated the new debugging analyses into the existing analysis pipeline of WatchDog. This pipeline extracts the data from WatchDog's Mongo database and loads it into R for further analysis. The analysis methods we used for some of these research questions require some more explanation.

For *RQ2.3*, we assessed the intervals that occur right before a debugging session is started. We chose a time period of 16 seconds as there is an inactivity timeout of 16 seconds in WatchDog, meaning that activity-based intervals like reading or typing intervals are automatically closed after this period of inactivity to account for e.g., coffee breaks.

For *RQ2.4* and *RQ2.5*, we consider a file "under debugging" if we receive reading or typing intervals during a debugging interval on it, i.e. for all the files the user steps through, reads, or otherwise modifies during a debugging session.

Study Subjects. Since the release of WatchDog 2.0 on 22 April 2016, we collected user data for a period over two months, until 28 June 2016. In this period, we received 1,155,189 intervals, from 458 users in 603 projects. Of these, 3,142 were debug intervals from 132 developers. In total, we recorded 18,156 hours in which the IDE was open,

which amounts to 10.3 observed developer years based on the 2015 average working hours for OECD countries [40]. We also collected 54,738 debugging events from 192 users, 218 projects and 723 IDE sessions. In total, we recorded both at least one debug interval and one event for 108 users. Table I presents the number of occurrences of the different event types.

B. Results

As Table I indicate, only 132 of the 458 users (28.8%) started a debugging session during the data collection period, with no significant difference between ECLIPSE (28.9%) and INTELLIJ (27.6%) users. Moreover, only 108 study subjects (23.6%) have used the debugger and at least one of its features (transferred both intervals and events).

WO1: The vast majority of WATCHDOG 2.0 users is not using the IDE-provided debugging infrastructure.

The results in Table I indicate that line breakpoints are by far the most used breakpoint type. The other, more advanced, types account for less than 7% of all breakpoints set during the collection period. Furthermore, line breakpoints are used by most developers using the debugging infrastructure, while the other types of breakpoints are used by only $7.6-20.5\%$ of these developers.

WO2: Line breakpoints are used most and by most developers, other breakpoint types are used much less and by much less developers.

The breakpoint change type frequencies in Table I indicate that almost all of these changes are related to the enablement or disablement of the breakpoints. The other change types account for only 10.9% of all breakpoint changes. Furthermore, the number of users that generated these events range from 1 (0.8%) to 12 (9.1%). Moreover, the events related to specifying a hit count on the breakpoint have not been recorded at all during the collection period.

WO3: Breakpoint options are not used by most WATCHDOG 2.0 users; the most frequently used option is changing their enablement.

Table I shows that most of the recorded events are related to the evolution of breakpoints, hitting them during debugging and stepping through the source code. The more advanced debugging features like defining watches and modifying variable values have been used much less. Furthermore, the same holds for the number of users generating these events: the majority of users have added and/or removed breakpoints and stepped through the code, while only $2.3-15.2\%$ modified variable values, evaluated expressions and/or defined watches.

WO4: Setting breakpoints and stepping through code is done most, other debugging features are used by far less.

For *RQ2.1*, we first computed the total duration of all intervals of a particular type and based it on the total duration of 'IDE open' intervals (18,156.9 hours, 100%) in the collection period. We recorded 25.2 hours of running unit tests (0.14%), 721.5 hours of debugging intervals (3.97%), 2,568.8 hours of reading (14.15%), and 1,228.6 hours of typing (6.77%). More generic interval types include e.g. 'WatchDog open' (0.12%) and 'IDE active' (28.92%). Next, we analyzed the duration and percentages on a per user basis. For the users with at least one debug interval, the descriptive statistics of the resulting duration and percentages are shown in Table II. This table also shows the corresponding values for reading, typing and JUNIT intervals.

From the results in Table II and the fact that the total recorded active IDE time was 5250.7 hours, we conclude that debugging consumes 13.7% of the total active in-IDE development time, while reading or writing code and running tests take 48.6%, 23.4% and 0.5%, respectively.

WO5: Debugging consumes, on average, less than 14% of the in-IDE development time.

For *RQ2.2* we are interested in knowing the frequency and length of debugging sessions. We first analyzed the number of debug intervals per user for the 132 developers that have used the debugger during the collection period. The resulting numbers range from a single debug interval to 598 debugging intervals, with an average of 23.8 and a median of 4 debug intervals per user. Next, we analyzed the duration of the 3,142 debug intervals and found values ranging from 3 milliseconds to 90.8 hours, with an average and median duration of 13.8 minutes and 42.3 seconds, respectively. About half of the users using the IDE-provided debugging infrastructure have launched the debugger four times or less during the two months of data collection, 21% launched their debugger more than 20 times.

WO6: About 20% of the developers are responsible for over 80% of the debugging intervals in our sample.

Furthermore, about half of the debugging sessions take at most 40 seconds, while about 12% of them last more than 10 minutes.

WO7: Most debugging sessions consume less than 10 minutes.

To find an answer to *RQ2.3*, we assessed the intervals that occur immediately before a debugging session starts. The resulting frequencies and their percentages of all intervals occurring before any debug interval are: 46 (0.48%) for running unit tests, 119 (1.24%) for other debug intervals, 4,991 (51.94%) for reading and 1,802 (18.75%) for typing intervals. About 70% of the debugging sessions start after reading or writing code; only 0.5% of them after a failing or passing test run.

WO8: Most debugging sessions start after reading or changing the code, not after running tests.

For *RQ2.4*, we researched whether there is a correlation between the file size of a class (in source lines of code [41]), and the number of times it is used for debugging. At $\rho = -0.75$, we find a strong negative correlation. We also investigated the relation between the file sizes and the duration of the debug intervals in which they are opened and found no significant correlation ($\rho = 0.19$).

TABLE I
FREQUENCY TABLES OF RECEIVED EVENTS AS WELL AS THE BREAKPOINT TYPES AND OPTIONS SEEN IN THEM.

Breakpoint type	Frequency	Breakpoint change type	Frequency	Event type	Frequency	Event type	Frequency
Class prepare	99	Change condition	3	Add breakpoint	4544	Resume client	8292
Exception	37	Disable condition	1	Change breakpoint	247	Suspend by breakpoint	13276
Field	78	Enable condition	19	Remove breakpoint	4362	Suspend by client	16
Line	4229	Disable	180	Define watch	343	Step into	3480
Method	77	Enable	40	Evaluate expression	101	Step over	19543
Undefined	24	Change suspend policy	4	Inspect variable	179	Step out	351
				Modify variable value	4		
	Σ 4544		Σ 247				Σ 54738

TABLE II
DESCRIPTIVE USAGE STATISTICS FOR KEY INTERVAL TYPES.

Variable	Unit	Min	25%	Median	Mean	75%	Max	Histogram
Debugging	Hours (%)	0.00 (0.00%)	0.03 (0.06%)	0.30 (0.49%)	5.47 (2.50%)	1.42 (2.36%)	333.70 (30.81%)	
Reading	Hours (%)	0.00 (0.02%)	0.14 (1.65%)	0.60 (3.22%)	5.70 (4.89%)	2.07 (5.68%)	591.10 (52.71%)	
Typing	Hours (%)	0.00 (0.01%)	0.21 (1.46%)	1.01 (3.59%)	2.95 (4.84%)	2.78 (6.87%)	63.87 (28.25%)	
Running JUNIT tests	Hours (%)	0.00 (0.00%)	0.00 (0.00%)	0.01 (0.01%)	0.68 (0.21%)	0.56 (0.16%)	9.19 (2.13%)	

WO9: Smaller classes are debugged more than larger classes.

For *RQ2.5* we aggregate and compared the number of classes in single debug intervals to: 1) the total number of classes we observed with WATCHDOG for this project (also through other intervals such as reading, writing, or running tests); and 2) the number of different classes that have been debugged during any debug interval of the project.

For 1), we found that on average only 4.83% (median: 1.66%) of all project classes we observed in WATCHDOG intervals were ever debugged. The value ranges from 0.22% to 100%, where the 100% cases possibly stem from toy projects with only one or two classes. For 2), the results range from 0.81% to 100% with an average of 14.47% (median: 4.55%). Both results seem to indicate that debugging is focused on a relatively small set of classes in the project. In 75% of debugging sessions, at most 5% of the project's classes are debugged.

WO10: In most cases less than 5% of the project's classes are debugged in a single debug session.

In *RQ2.6*, we first investigated the relation between the total duration of running unit tests and debug intervals per user. We only considered the 25 developers with at least one debug interval and one unit test execution. At $\rho = 0.58$, we find a moderate correlation between the two durations.

WO11: Developers who spend more time executing tests are likely to proportionally debug more.

Next, we studied the relation between the amount of time the user spends inside test classes and the debugging time. For the 248 developers with at least one debug interval or one opened test class, we find no correlation at $\rho = -0.08$. Furthermore, we find no significant correlation ($\rho = 0.23$) when focusing on the 84 users with both at least one debug interval and one opened test class.

WO12: Developers who read or modify test classes longer are not significantly likely to debug less.

For answering *RQ2.7*, we first computed the total duration of all debug intervals per user. Then, we performed a Spearman rank-order correlation test using these values and the programming experience the user entered during WATCHDOG 2.0's registration process by applying a linear integer transformation (see Section III-B). For the 58 users that have entered their experience and generated at least one debug interval, this resulted in a weak correlation ($\rho = 0.38$).

WO13: More experienced developers are likely to spend slightly more time in the IDE debugger.

During our research into debugging, we sometimes heard anecdotal reports of frustrated developers stepping over the point of interest while debugging. To this end, we sought objective data to support how severe of a problem it really is by looking for possible cases of stepping over the point of interest for *RQ2.8*, which means that the developer steps one time too many and has to start debugging all over again. For this we created a set of debug intervals that satisfy the following two conditions: 1) the last event occurring within the debug interval is a stepping event; and 2) the interval is followed by another debug interval in the same IDE session. Then we created several subsets of this debug intervals by imposing a maximum time period between two consecutive debug intervals. Figure 6 shows the possible cases of stepping over the point of interest for the subsets with a maximum time period of 15 minutes.

The trend line in Figure 6 shows that the amount of new possible cases of stepping over the point of interest starts to decrease significantly after about four minutes. At this point, about 150 possible overshoot cases can be identified, which corresponds to 4.8% of the debugging intervals.

WO14: Developers might step over the point of interest and have to start over again in 5% of debugging sessions.

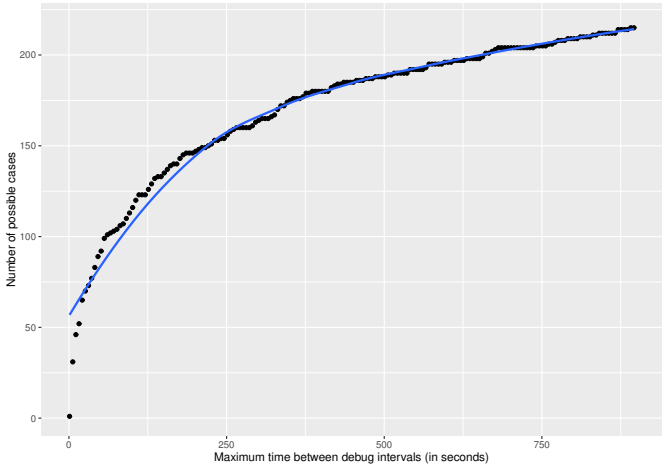

Fig. 6. Possible cases of stepping over the point of interest per maximum time period between consecutive debug intervals.

TABLE III
INTERVIEWED DEVELOPERS AND DEBUGGING EXPERTS

ID	Occupation	Dev. Experience	Country	Area
I1	Freelancer	>20 years	Germany	Rich Client Platforms
I2	Developer	≥ 15 years	India	E-commerce
I3	Developer	11 years	USA	Real-Time Systems
I4	Developer	10 years	UK	Data Scraping
E1	3 Eclipse Debugging Projects Leads		Switzerland, India	Eclipse Development
E2	Professor	>20 years	Greece	Software Engineering
E3	Debugger Developer	18 years	Russia	IDE Development

V. INTERVIEWS

In this section, we describe how we conducted developer interviews for *RQ3* and merge and discuss results from *RQ1* and *RQ2*.

A. Study Methods

Interview Design & Method. To validate and obtain a deeper understanding of our findings from *RQ1* and *RQ2* and to mitigate apparent controversies, we performed two sets of interviews. First, we sampled a set of "regular developers" from our survey population to gain insights into what hinders the use of debuggers, why printf debugging is still widely used, and whether they regularly step over the line of interest. Then we ran the combined observations from survey, objective IDE measurement, and anecdotal interview insights across three sets of debugging experts. A question sheet helped us steer the semi-structured interviews, which we conducted remotely via Skype and took from 36 minutes to one hour and seven minutes. Subsequently, we transcribed the interviews to text and extracted insightful quotes.

Study Subjects. Table III gives an overview of our study population of nine interviewees. In one case (*E3*), we performed the interview asynchronously via email.

B. Results

This section juxtaposes survey (*RQ1*) and IDE study (*RQ2*) results and discusses them by the help of the qualitative insights we obtain from *RQ3*.

Use of the IDE Debugger. In *WO1*, we found that two thirds of the WATCHDOG 2.0 users were not using the IDE-provided debugger in our observation period, an obvious contradiction to *SO1,* in which 80% of respondents claimed to use it. There might be several reasons for the discrepancy: 1) The study populations are different, and the survey respondents were likely self-selecting on their interest in debugging, resulting in a higher than real use of the debugger. 2) As often observed in user studies, most relevant data stems from a relatively small percentage of users. 3) WATCHDOG users were free to start and stop using the plugin at any time in the observation period. Hence, for some users the actual observation period might be much shorter, perhaps coinciding with not having to debug a problem. 4) Almost equally many developers conceded to use printf statements for debugging. We have anecdotal evidence from *RQ3* that they might use them even more: When we asked *I3* about printf debugging, he was very negative about it. Later in the interview, he still conceded to use printf "very rarely." We believe a similar case might hold for many WATCHDOG users. As we cannot capture printf debugging or debugging outside the IDE with WATCHDOG, our finding does not mean two thirds of developers did not debug. 5) The phenomenon of a discrepancy between survey answers and observed behavior is not new. We observed it similarly with developers claiming to test, and then not nearly testing as much in reality [37]. As a consequence, we emphasize our previous finding that survey answers always be cross-validated by other methods.

Printf Debugging. From *RQ1* and *RQ2*, it seemed that developers were well-informed about printf debugging and that it is a conscious choice if they employ it, often the beginning of a longer debugging process. Interviewees praised printf as a universal tool that one can always resort back to, helpful when learning a new language ecosystem, in which one is not yet familiar with the tools of the trade. About left-over print statements that escape to production, I2 was "not worried at all, because we have a rigorous code review process." While frequently used, developers are also aware of its shortcomings, saying that "you are half-assing [sic!] your way toward either telemetry or toward tracing" and "that it is insufficient for concurrent programs, primarily because the [output] interleave[s] in strange ways" (*I3*).

Use of Debugging Features. *SO3* and *SO4* indicated that most developers use line breakpoints, but do not use more advanced breakpoint types like class prepare breakpoints. While many developers knew and used conditional breakpoints, they were widely ignorant of hit counts and more advanced functions of the debugger. *WO2* to *WO4* support this result, finding that conditional breakpoints are indeed the second most feature in the IDE debugger. A similar result is visible in other debugging features like stepping through code. In both cases we found that these features get used less as they become more advanced. However, the observed numbers on the use of these features are much lower than the claimed usage visualized in Figure 2. For example, while 60% of the survey respondents indicated to define watches during debugging, only 15.2% of the WATCHDOG 2.0 users that use the debugger have defined

a watch. Through our interviews with the debugging experts, we identify three possible causes for this.

1) More advanced debugging features are seldom required. *I1* and *I2* said that specifying conditions or hit counts is often "fuzzy" (is it going to happen on the 16th, 17th, or 18th time?) and that once one knows the condition, one almost automatically understands the problem, and then there is no need for the conditional breakpoint anymore. Moreover, "the types of problems where you need a conditional breakpoint happen very, very rarely" (*I2*). For example, when we presented the breakpoint export feature of ECLIPSE to *I2*, he replied "oh, I did not know such a feature exists." Others said it is a "very esoteric thing" and that they have used it "maybe once or twice" (*I3*). This strengthens our intuition that debugging is a process that is usually not shared with the outside and that breakpoints are "like a one-shot. Ideally I wouldn't like them to be, but then I just set them anew" (*I4*).

2) The debugger is difficult to use. Another reason given by interviewees, even though seasoned engineers, was that "the debugger is a complicated beast" (*I2*) and that "debuggers that are available now are certainly not friendly tools and they don't lend toward self-exploration." Given our results on the use of features, we asked interviewees whether it might simply be enough to reduce the feature set. Both developers and *E1* to *E3* emphatically declined, arguing that "once you get into these crazy cases, they are really useful" (*I2*).

3) There is a lack of knowledge on how to use the debugger. When we asked developers where their knowledge of debugging comes from, many said that "big chunks are self-taught" and "[I] picked up various bits and pieces on the Internet" (*I4*). Even *I3*, the only interviewee who indicated that "debugging was explicitly covered [in my undergraduate]," said it is "partly self taught, partly [...] through key mentorships." Making a case for hands-on teaching, he elaborated that "one of the engineers that mentored me [...] was some kind of wizard with gdb. I think when you meet someone who knows a very powerful tool it's very impressive and their speed to resolving something is much faster but it takes a lot of time to get to that point." Since we measured experience to have limited to no impact on (which) debugging features developers used (*SO6*), this hints at a lack of education on debugging that is pervasive from beginners and Computer Science students to experts. New Computer Science curricula that put debugging upfront could be an effective way to steer against it.

Time Effort for Debugging. Our study results *WO5* to *WO7* point to the fact that debugging in most cases is a short, "get-it-done" (*I1*) type of activity that, with only 14% of active IDE time (*WO5*) we found to consume significantly less than the $30-90\%$ for testing and debugging reported by Beizer [42] and than the estimations by our interviewees, who gave a range of 20% to 60% of their active work time. One reason why our measured range is so much lower is that developers (and humans in general) seem to have a tendency to overestimate the duration of unpleasant tasks, as we previously observed with testing [37]. Another might be that developers included debugging tasks such as printf and the use of external tools, which we cannot measure. We need more studies to quantify this initial surprising finding.

Use of Tests for Debugging. In the survey, most respondents think (unit) testing is an integral part of the debugging process, especially for reproducing bugs at the beginning of the process (*SO7*). However, there is mixed evidence on this in *RQ2*, as shown by *WO8, WO11* and *WO12*. On the one hand, failing tests do not seem to be a trigger for the start of debugging sessions. On the other hand, running tests in the IDE seems to be correlated with debugging more, while reading or modifying tests is not. Two factors can play a role: Developers who are more quality concerned, execute their tests more often and therefore also debug more. This is contrary to intuition and the answers of some of our interviewees, who claimed that as testing goes up, the debugging effort should decrease (*E2*): "debugging is born of unknowns, and effective testing reduces these" (*I3*). An explanatory finding might be that the creation of tests itself adds code and complexity that might need to be debugged. We need more studies to research this surprising discovery.

Stepping Over the Point of Interest. We found that in less than about 5% of the debugging sessions the developer might have stepped over the point of interest and had to start debugging anew (*WO14*). This indicates that there is a limited, but existent gap in current debuggers process that might be filled by *back-in-time debuggers* [25]. Back-in-time debugger allow developers to step back in the program execution in order to arrive at the point of interest without having to completely restart the debugging process. All our interviewees could relate to situations in which this occurred to them, stating that "it happens all the time" (*I1*) to "back in time debugger would be wonderful" (*I3*). However, *WO14* indicates that it might not be as frequent as some stated. While the drop frame feature allows developers to go to the beginning of the current method, it does not revoke side effects that already occurred and was therefore found to be "helpful in a limited way" (*I3*). Currently, mainstream IDEs do not support back-in-time debugging.

Improvement Wishes. We asked our interviewees how debugger creators could better support them. Their answers fall into two categories: 1) Make the core features easier to use while pertaining all existing functionality. 2) Create tools that capture the holistic debugging process better. We elaborate here on 2). *I1* denotes: "If you're in Java and have to debug across language boundaries, [...] you really get to a point where you feel helpless." Other wishes included the ability to do back-in-time debugging similar to CHRONON [43], to have a live REPL, a feature the IDE XCODE introduced [44].

To possibly inform the design of future IDE debuggers, we arranged a meeting with three debugging project leads from ECLIPSE, *E1*, and an IDE developer from a commercial company, *E2*. The ECLIPSE leads said that, while they know how individual developers use their debugger, they are unaware of the number and detail our study could provide. They thanked us "for all [...] suggestions made during the call, they were really useful" and started or updated six feature requests for

the debugger based on our study.[3]

C. Threats to Validity

In this section, we examine threats to the validity of our study and show how we mitigated them.

Construct Validity. The manual implementation of new functionality, such as the addition of the debug infrastructure to WATCHDOG, is prone to human errors. To minimize these risks, we extended WATCHDOG's automated test suite. Furthermore, we use this test suite to make sure we introduced no regressions. In addition, we tested our plugins manually. Finally, we performed rigorous code reviews before we integrated the changes. Debug sessions might not correspond to actual debug work, e.g. a user might have inadvertently left the debugger in the IDE running, explaining our 90 hour outlier. Similarly, we approximate the number of classes in a project by the number of different classes we observe with WATCHDOG. Due to privacy reasons, we cannot mine the repositories of projects to gain an entirely correct figure. However, when not considering extreme outliers, the wealth of our data looks very plausible, strengthening confidence in our study setup.

Internal Validity. Since our survey in *RQ1* dealt with debugging, participation might have been self-selecting, i.e. developers more interested and knowledgeable in debugging are more likely to have responded. We tried to contrast this with objective WATCHDOG observations, which is not advertised specifically as a debugging tool. An important internal threat is that the populations for *RQ1* and *RQ2* are different and their intersection small (six users participated in both studies). However, we are confident we only encounter a small sampling or comparison bias because key characteristics of both populations are similar, as 1) 80% of respondents answered the survey for Java, which both plugins work with in *RQ2*, 2) the majority in *RQ1* used one of the IDEs supported in *RQ2*, 3) the experience distributions of both populations are similar and 4) both populations should be large enough to even out individual influences.

External Validity. During our data collection period of more than two months we collected 1,155,189 intervals with a total duration of over ten developer years, spread over 458 users. The fact that over 80% of the survey respondents stem from the Java community means that little survey data is available about other communities. The same holds for the analysis of the WATCHDOG 2.0 data, which is restricted to the Java programming language and to the ECLIPSE and INTELLIJ IDEs. Other IDEs are not included in our analysis and the results with them might deviate. However, at least imperative, statically typed languages similar to Java, like C, C++, C#, or Objective-C, would likely yield similar results and are so widespread that researching them alone impacts many, if not the majority of, developers.

[3]The umbrella bug 498469 describes and links to the sub change requests, see https://bugs.eclipse.org/bugs/show_bug.cgi?id=498469.

VI. Future Work & Conclusion

Based on the insights obtained from the online survey, the fine-grained debugging activity data originating from WATCHDOG 2.0 and the interviews with professional software developers and debugging experts, we have formulated several conclusions that have already impacted both education and practice, for example, the introductory course on Computer Science education at TU Delft, which previously contained no lecture on debugging. We believe that more educators can follow this example and include practical, hands-on teaching on debugging. It is puzzling that we know that bugs are inevitably linked with software and that we teach students to write software from day one, but only give them the knowledge to properly debug into these much later, if ever.

Apart from this lack in education, we also found that debuggers are not easy to use, and, with the help of three ECLIPSE project leads, identified several concrete areas of improvement in the ECLIPSE debugger, leading to several feature and change requests. Other debugging tool creators could follow this example and use our findings to improve their debuggers.

As researchers, to strengthen the generalizability of some of our results, we plan to collect WATCHDOG 2.0 data over a longer period of time. To overcome the limitation of only collecting data on ECLIPSE and INTELLIJ and therefore mainly on Java developers, we aim to support more non-Java IDEs through our INTELLIJ plug-in family approach [39]. Finally, to better compare developers' perception on debugging and their behavior, we setup the infrastructure which links survey responses to WATCHDOG 2.0 data, allowing research on the same population. However, at the time of writing, too few responses were available to draw significant results from this study-worthy sub-population. We plan to follow-up with a larger population and also conduct interviews with WATCHDOG users.

The key contributions of this paper are:

- The creation of WATCHDOG 2.0, an OSS multi-platform infrastructure that allows detailed tracking of developers' debugging behavior.[4]
- A triangulated, large-scale empirical study of how developers debug in reality using a mixed methods approach.
- Improvement suggestions for Computer Science curricula and current IDE Debuggers that have in part already been implemented in practice.

We currently research ways to offer a live replication package.[5]

Acknowledgments

We thank all study participants who in spite of showing their fallibility allowed us to do research on their debugging behavior. In particular, we thank all developers who contributed to our survey, sent us WATCHDOG data, and dedicated their time and energy to very insightful and joyful interviews.

[4]https://testroots.org/testroots_watchdog.html

[5]Made available in case of acceptance

REFERENCES

[1] D. Spinellis, *Effective Debugging: 66 Specific Ways to Debug Software and Systems*. Addison-Wesley, 2016.

[22] M. Perscheid and R. Hirschfeld, "Follow the path: Debugging tools for test-driven fault navigation," in *Software Maintenance, Reengineering*

[2] Q. Zhao, R. Rabbah, S. Amarasinghe, L. Rudolph, and W.-F. Wong, "How to do a million watchpoints: efficient debugging using dynamic instrumentation," in *Proceedings of the Joint European Conferences on Theory and Practice of Software and 17th international conference on Compiler construction (CC'08/ETAPS'08)*. Springer, 2008, pp. 147–162.

[3] A. Zeller, *Why Programs Fail, Second Edition: A Guide to Systematic Debugging*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009.

[4] B. Siegmund, M. Perscheid, M. Taeumel, and R. Hirschfeld, "Studying the advancement in debugging practice of professional software developers," in *Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on*. IEEE, 2014, pp. 269–274.

[5] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, pp. 11:1–11:32, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2000791.2000795

[6] S. Chandra, E. Torlak, S. Barman, and R. Bodik, "Angelic debugging," in *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 2011, pp. 121–130.

[7] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ser. ISSTA '10. New York, NY, USA: ACM, 2010, pp. 61–72. [Online]. Available: http://doi.acm.org/10.1145/1831708.1831716

[8] M. Z. Malik, J. H. Siddiqi, and S. Khurshid, "Constraint-based program debugging using data structure repair," in *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*. IEEE, 2011, pp. 190–199.

[9] 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.

[10] G. Tassey, "Economic impacts of inadequate infrastructure for software testing, planning report 02-3," *Prepared by RTI for the National Institute of Standards and Technology (NIST)*, 2002.

[11] R. Stallman, R. Pesch, S. Shebs *et al.*, *Debugging with GDB*, 10th ed. Free Software Foundation, 2011.

[12] E. Burnette, *Eclipse IDE Pocket Guide*. " O'Reilly Media, Inc.", 2005.

[13] A. Zeller and D. Lütkehaus, "Ddda free graphical front-end for unix debuggers," *ACM Sigplan Notices*, vol. 31, no. 1, pp. 22–27, 1996.

[14] A. Zeller, "Isolating cause-effect chains from computer programs," in *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*. ACM, 2002, pp. 1–10.

[15] F. Eichinger, K. Krogmann, R. Klug, and K. Böhm, "Software-defect localisation by mining dataflow-enabled call graphs," in *Machine Learning and Knowledge Discovery in Databases*. Springer, 2010, pp. 425–441.

[16] S. Parsa, M. Vahidi-Asl, S. Arabi, and B. Minaei-Bidgoli, "Software fault localization using elastic net: A new statistical approach," in *Advances in Software Engineering*. Springer, 2009, pp. 127–134.

[17] D. Abramson, C. Chu, D. Kurniawan, and A. Searle, "Relative debugging in an integrated development environment," *Software: Practice and Experience*, vol. 39, no. 14, pp. 1157–1183, 2009. [Online]. Available: http://dx.doi.org/10.1002/spe.932

[18] C. Zhang, J. Yang, D. Yan, S. Yang, and Y. Chen, "Automated breakpoint generation for debugging," *Journal of Software*, vol. 8, no. 3, 2013. [Online]. Available: http://www.ojs.academypublisher.com/index.php/jsw/article/view/jsw0803603616

[19] J. Rößler, G. Fraser, A. Zeller, and A. Orso, "Isolating failure causes through test case generation," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. New York, NY, USA: ACM, 2012, pp. 309–319. [Online]. Available: http://doi.acm.org/10.1145/2338965.2336790

[20] Y. Lei, X. Mao, Z. Dai, and C. Wang, "Effective statistical fault localization using program slices," in *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual*, July 2012, pp. 1–10.

[21] S. Parsa, F. Zareie, and M. Vahidi-Asl, "Fuzzy clustering the backward dynamic slices of programs to identify the origins of failure," in *Experimental Algorithms*. Springer, 2011, pp. 352–363.

and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on. IEEE, 2014, pp. 446–449.

[23] K. Yu, M. Lin, J. Chen, and X. Zhang, "Practical isolation of failure-inducing changes for debugging regression faults," in *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*. IEEE, 2012, pp. 20–29.

[24] A. Orso, "Automated debugging: Are we there yet?" https://www.youtube.com/watch?v=WJHQnzLpVXk&feature=youtu.be, 2014, [Online; accessed 11 July 2016].

[25] G. Pothier and É. Tanter, "Back to the future: Omniscient debugging," *IEEE software*, vol. 26, no. 6, pp. 78–85, 2009.

[26] M. Perscheid, B. Siegmund, M. Taeumel, and R. Hirschfeld, "Studying the advancement in debugging practice of professional software developers," *Software Quality Journal*, pp. 1–28. [Online]. Available: http://dx.doi.org/10.1007/s11219-015-9294-2

[27] D. Piorkowski, S. D. Fleming, C. Scaffidi, M. Burnett, I. Kwan, A. Z. Henley, J. Macbeth, C. Hill, and A. Horvath, "To fix or to learn? how production bias affects developers' information foraging during debugging," in *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, ser. ICSME '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 11–20. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2015.7332447

[28] D. J. Piorkowski, S. D. Fleming, I. Kwan, M. M. Burnett, C. Scaffidi, R. K. Bellamy, and J. Jordahl, "The whats and hows of programmers' foraging diets," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2013, pp. 3063–3072.

[29] F. Petrillo, Z. Soh, F. Khomh, M. Pimenta, C. Freitas, and Y.-G. Guhneuc, "Understanding interactive debugging with swarm debug infrastructure," in *Proceedings of the 24th International Conference on Program Comprehension*. ACM, 2016, pp. 1–4.

[30] M. Das, P. Ester, and L. Kaczmirek, *Social and behavioral research and the internet: Advances in applied methods and research strategies*. Routledge, 2010.

[31] D. Spencer, *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.

[32] P. E. Greenwood and M. S. Nikulin, *A guide to chi-squared testing*. John Wiley & Sons, 1996, vol. 280.

[33] J. L. Devore and N. Farnum, *Applied Statistics for Engineers and Scientists*. Duxbury, 1999.

[34] W. G. Hopkins, *A new view of statistics*, 1997, http://newstatsi.org, Accessed 16 March 2015.

[35] T. V. Perneger, "What's wrong with bonferroni adjustments," *Bmj*, vol. 316, no. 7139, pp. 1236–1238, 1998.

[36] D. A. Keim, "Visual exploration of large data sets," *Commun. ACM*, vol. 44, no. 8, pp. 38–44, 2001. [Online]. Available: http://doi.acm.org/10.1145/381641.381656

[37] M. Beller, G. Gousios, A. Panichella, and A. Zaidman, "When, how, and why developers (do not) test in their ides," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 179–190. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786843

[38] M. Beller, G. Gousios, and A. Zaidman, "How (much) do developers test?" in *Proceedings of the 37th International Conference on Software Engineering (ICSE), NIER Track*. IEEE, 2015, pp. 559–562.

[39] M. Beller, I. Levaja, A. Panichella, G. Gousios, and A. Zaidman, "How to catch 'em all: Watchdog, a family of ide plug-ins to assess testing," in *Proceedings of the 3rd International Workshop on Software Engineering Research and Industrial Practice*, ser. SER&IP '16. New York, NY, USA: ACM, 2016, pp. 53–56. [Online]. Available: http://doi.acm.org/10.1145/2897022.2897027

[40] O. for Economic Co-Operation and Development, "Average annual hours actually worked per worker," http://stats.oecd.org/index.aspx?DataSetCode=ANHRS, 2015, [Online; accessed 11 July 2016].

[41] V. Nguyen, S. Deeds-Rubin, T. Tan, and B. Boehm, "A sloc counting standard," in *COCOMO II Forum*, vol. 2007, 2007.

[42] B. Beizer, "Software testing techniques. 1990," *New York, ISBN: 0-442-20672-0*.

[43] C. Systems, "Chronon, a dvr for java," http://chrononsystems.com/, 2015, [Online; accessed 24 August 2016].

[44] A. Inc., "Xcode 8," https://developer.apple.com/xcode/, 2015, [Online; accessed 24 August 2016].