Delft University of Technology

On the Dichotomy of Debugging Behavior Among Programmers

Beller, Moritz; Spruit, Niels; Zaidman, Andy; Spinellis, Diomidis

On the Dichotomy of Debugging Behavior Among Programmers

Moritz Beller
Delft University of Technology
The Netherlands
m.m.beller@tudelft.nl

Niels Spruit
Delft University of Technology
The Netherlands

Diomidis Spinellis
Delft University of Technology
The Netherlands

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

ABSTRACT

Debugging is an inevitable activity in most software projects, 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 on how 458 developers use the debugger included in their integrated development environments (IDEs) by instrumenting the popular ECLIPSE and INTELLIJ IDEs with the purpose-built plugin WATCHDOG 2.0. To clarify the insights and discrepancies observed in the previous steps, we followed up by conducting interviews with debugging experts and regular debugging users. Our results indicate that IDE-provided debuggers are not used as often as expected, because "printf debugging" remains a feasible choice for many programmers. Furthermore, both knowledge and use of advanced debugging features are low. Our results call for strengthening hands-on debugging experience in computer science curricula and have already refined the implementation of modern IDE debuggers.

CCS CONCEPTS

• **Software and its engineering** → Software testing and debugging;

1 INTRODUCTION

Debugging, the activity of identifying and fixing faults in software [1], is a tedious, but inevitable activity in almost every software development project [2]. Not only is it inevitable, but according to Kernighan and Plauger [3] and Zeller [4], so difficult that it often consumes more time than creating the bogus piece of software in the first place.

During debugging, software engineers need to relate an observed failure to its underlying defect [5]. To complete this step efficiently, they often need to acquire a deep understanding and build a mental model of the software system at hand [6]. This is where modern debuggers come in: they aid software engineers in gathering observing the system's dynamic behavior. However, they still require them to select the parts on which to focus and to perform the deductive reasoning to pinpoint the fault from the observed behaviors.

While the scientific literature is rich in terms of proposals for (automated) debugging approaches, e.g., [4, 7–10], there is a gap in knowledge of how practitioners actually debug. Debugging has thus remained *the dirty little secret of computer science* [11]. How and how much do software engineers debug at all? Do they use modern debuggers? Are they familiar with their capabilities? Which other tools and strategies do they know?

The lack of knowledge regarding 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 precarious situation for both a developer and a researcher. Nevertheless, by continuing to keep debugging practices secret, we miss an important opportunity for advancing software engineering theory and for delivering efficiency improvements in software development practice.

Knowledge on how developers debug can help researchers to invent more practice-relevant techniques, educators to improve their debugging curricula, and tool builders to tailor debuggers to the actual needs of developers. To open up the art of debugging, we conducted a large-scale behavioral field study on what developers think about debugging and how they debug in their IDEs. The following main questions steer our research:

RQ1 What do developers know about debugging and how do they reflect on it?

RQ2 How do developers debug in their IDEs?

RQ3 How do individual debugger users and experts interpret our findings from *RQ1* and *RQ2*?
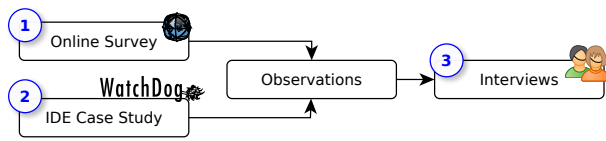
Figure 1: Research design overview.

The key contributions of this paper are:
- A triangulated, large-scale empirical study of how developers debug in reality using a mixed methods approach.
- The addition of debugging features in WATCHDOG 2.0, an open-source, multi-platform infrastructure that allows detailed tracking of developers' debugging behavior.[1]
- Improvement suggestions for current IDE debuggers that have in part already been implemented in practice.

Research Design

To answer these research questions, we employed a multi-faceted research approach outlined in Figure 1. ① We conducted an online survey to capture developers' opinions on debugging and obtained an overview of the state of the practice (see Section 3, Survey Results, SR). ② Simultaneously, we began using the automated WATCHDOG 2.0 infrastructure to track developers' fine-grained debugging activities in the IDE (see Section 4, WATCHDOG Results, WR). By instrumenting the IDE, we obtained objectively measured data, which we can compare against subjective, but richer data from the survey. We came up with a list of several, sometimes conflicting, observations that needed further explanation. ③ To help us explain the findings in depth, we conducted interviews with developers, some of whom are actively developing debugging tools (see Section 5, Interviews).

2 RELATED WORK

Work related to our study comprises debugging tools, processes, techniques, empirical debugging evidence, and IDE instrumentation.
Debugging Tools. By "debuggers," we usually mean *symbolic debuggers*, such as the GNU Project Debugger (GDB) [12]. These debuggers allow developers to specify points in the program where the execution should halt, *breakpoints*. A typical symbolic debugger supports different *types of breakpoints*, such as *line, method, data access*, or more advanced *exception* or *class prepare breakpoints*, and options to refine the breakpoint [13]. Examples include specifying a *conditional breakpoint*, a *hit count*, a *suspension policy*, or whether the entire program or one thread should pause upon hitting a breakpoint.

Once a program halts, developers can use the symbolic debugger to permanently *watch* or ad-hoc *inspect* memory entities such as variables, work through the call stack, line-wise step through the code, or evaluate arbitrary expressions [12, 13]. Graphical debuggers like the early *dbxtool* [14] and DDD [15] evolved from command line symbolic debuggers, such as VAX DEBUG [16], *dbx* [17], and GDB [18]. Most symbolic debugging features have since been integrated in the integrated graphical debuggers of IDEs, such as ECLIPSE, Visual Studio, NetBeans, and INTELLIJ. This study focuses on how developers use IDE debuggers.

Debugging Process. Researchers have developed systematic process descriptions of debugging and recommendations to reduce the time programmers have to spend on finding and fixing a defect that causes a program failure. We investigate whether developers explicitly or implicitly use debugging strategies inspired by the scientific method; for example, Zeller's *TRAFFIC* approach [4] 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 the steps regard "the 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 [19].

In 1991, Gilmore suggested a new psychological model to understanding debugging [20]. Component 1 of his model, namely that debugging is a "flexible, incomplete comprehension process [...] according to task demands, tools and skill," provides a theory-grounded description of our observations.
Automated 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 [21]. Other types of debugging technique include *slicing* [4], focusing on *anomalies* [4], *mining dynamic call graphs* [22], *statistical debugging* [23], *spectra-based fault localization* [7], *angelic debugging* [8], *data structure repair* [10], *relative debugging* [24], *automatic breakpoint generation* [25], *automatic program fixing using contracts* [9], and combinations thereof [26–30]. Orso presents a detailed overview of some of these automated debugging techniques [31]. However, as our study shows, automated debugging techniques have not yet reached the mainstream debugging practices and are not part of IDE debuggers. As such, we do not discuss them further.
Empirical Debugging Evidence. To the best of our knowledge, only few studies exist that empirically evaluate how developers debug. In a general 2006 study on how developers use Eclipse, Murphy et al. found that 90% of their 41 studied developers used the debugger [32]. This is similar to our debugger use rate in *WR1* when only considering the top 10% of users. Parnin and Rugaber observed that 13% of their 10,000 recorded sessions included debugging, compared to an IDE debugger use in 9% of the sessions in our study [33] (*WR1*).

Siegmund et al. studied debugging practices of professional software developers [5]. While their research effort is perhaps closest to ours, it differs 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 described in TRAFFIC [4]. "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 [34], 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,

---
[1] <identifying link removed>

approach and education" [35]. 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." As in our survey, concurrency issues and external libraries seem to be the root causes of the hardest bugs.

Despite differences in study populations and methods, Layman et al. found similar challenges and improvement wishes such as concurrency (*SQ13*) and back-in-time debugging [36]. However, they do not mention some of the critical challenges found in this paper, such as debugging across languages.

Piorkowski et al. studied qualitatively how programmers forage for information [37, 38]. They found that developers spent half of their debugging time foraging for information. This complements our study as it shows what parts of the IDE are often used for finding information during debugging.

Böhme et al. studied individual steps in the debugging process, i.e., how developers localize, diagnose, and fix faults [39, 40]. Through an experiment with 12 professional software engineers they observed that fault localization is complex due to errors from an interactions of several statements. They also found that participants diagnosed bugs in a remarkably similar way. However, when fixing a fault, while the patches submitted by the participants were plausible, only 58% were correct.

IDE Instrumentation. Petrillo et al. developed the *Swarm Debug Infrastructure* (SDI), which "provides [Eclipse] tools for collecting, sharing, and retrieving debugging data" [41]. Developers can use the collective knowledge of previous debug sessions to "navigate sequences of invocation methods" and "find suitable breakpoints." SDI was evaluated in a controlled experiment involving 10 developers. Our ECLIPSE instrumentation for *RQ2* is technically similar to SDI, but focuses on understanding current behavior. To increase generalizability, we also support INTELLIJ and performed a longitudinal study of how dozens of developers debug in the wild.

While several WATCHDOG-like plugins for IDE-instrumentation exist [42–45], none of them have been used to study the debugging behavior of developers, manifesting our knowledge gap of empirical debugging. Ko and Myers showed the practical usefulness of the "live IDE" wish expressed in *SQ13* with their Whyline prototype [19], which helps developers reason about assumed program behavior.

3 DEBUGGING SURVEY

In this section, we describe our online survey.

3.1 Research Methods

Survey Design. To investigate developers' self-assessed knowledge on debugging for *RQ1*, we set up a survey.[2] 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

---
[2] <identifying link removed>

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 about participants' opinion on the statement "the best invention in debugging was printf debugging," inspired by Brian Kernighan's quote "[t]he most effective debugging tool is still careful thought, coupled with judiciously placed print statements" [46, 47]. We included the question because research on survey design has shown that posing a concrete, controversial statement that evokes strong opinions leads to more insightful answers [48]. Before publicly releasing the survey, we sharpened it in several iterations and ran six trials with outsiders.

Card Sort. To gain an overview of the topics that concern developers, we performed an *open card sort* [49] on *SQ13*. Two authors individually built and then mutually agreed on a set of 33 tags from a sub-sample of responses. After labeling all responses (possibly with multiple labels), a third author sampled 20% of the tagged responses, re-tagged them independently and compared them to the reference tag set. We then converged our tag sets into a homogeneous classification with 34 tags, agreed upon by all authors.

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 pairwise *Pearson Chi-Squared (χ^2) test of independence* [50], 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 [51]. For interpreting the results of dependency analyses ρ, we use Hopkins' guidelines [52]. They call $0 \leq |\rho| < 0.3$ no, $0.3 \leq |\rho| < 0.5$ weak, $0.5 \leq |\rho| < 0.7$ moderate and $0.7 \leq |\rho| \leq 1$ a strong correlation.

Subject Recruitment. To attract survey participants (*SP*), 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%). We asked for the language to understand whether we can compare the survey results to our Java-based field study, and because certain language features define their debugging possibilities, for example the availability of a virtual machine in Java [53] or Pharo's introspection design concept, which lends itself to debugging [54, 55].

3.2 Results

Analysis of Survey Answers. In this section, we describe key results of our survey and RQ1.

SR1: Most developers use IDE debuggers in conjunction with log files and print statements. In our first question, 143 developers (81.3%) indicated that they use the IDE-provided debugging infrastructure, 15 (8.5%) that they do not, and 18 (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, internal or non-generalizable 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).

SR2: Developers not using the IDE debugging find external programs, tests, print statements, or other techniques more effective or efficient. Of the 15 developers not using the debugging infrastructure, eight think that print statements and six that techniques other than print statements are more effective or efficient. Six use an external program, while four do not know how to use a debugger.

SR3: Line breakpoints are used by the vast majority of developers. More advanced types are unknown to most. 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. 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.

SR4: Most developers answered to be familiar with breakpoint conditions, but not with hit counts and suspend policies. 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. 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 [5]. The more advanced features, like defining watches or a suspend policy, seem to be known and used less.

SR5: Survey answers indicate testing is an integral part of the debugging process, especially at the beginning and end. Figure 3 shows 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, but slightly less during the process.

SR6: Experience has limited to no impact on the usage of the IDE-provided debugging infrastructure and tests. Examining our survey answers for dependencies allows us to understand how certain answers 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. We find that there is no correlation between the use of an IDE debugger or (unit) tests for debugging and experience in software development.
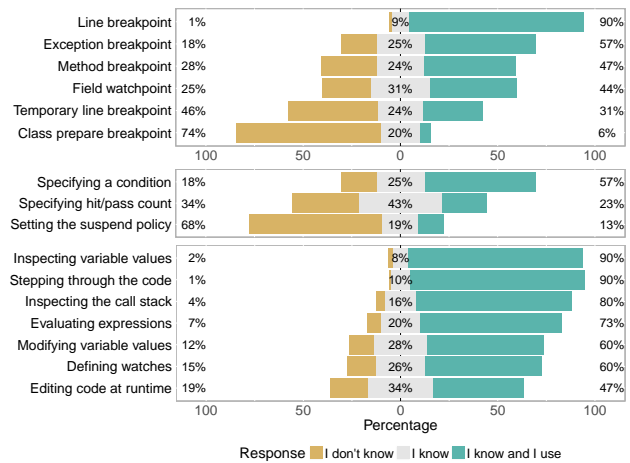

Figure 2: Answers in *SQ7–9* on breakpoint types, breakpoint options, and debugging features (*n* = 143).
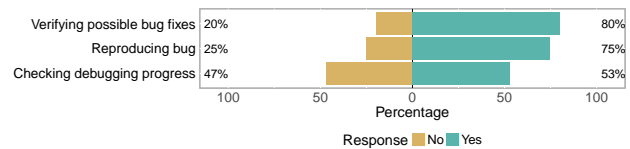

Figure 3: Answers in *SQ10–12* on unit tests (*n* = 176).

There is a weak correlation between experience and specifying hit counts and a moderate correlation between experience and the usage of watches during debugging.

SR7: 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. 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.

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 interpretation 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 very low normalized weight (cleaning the graph from "background noise"), and (3) we removed vertices that did not have any outgoing or incoming edge (removing unrelated concepts). The resulting graph in Figure 4 allows us an intuitive understanding and overview of responses and how they relate to each other, without having to read hundreds of responses [56].[3] The tags abbreviate concepts given as answers by survey respondents and are

---

[3]We explicitly avoided statistical tests. Given open-ended survey answers, the meaning of such tests is unclear, or might convey a false sense of statistical precision at worst. The graph conveys our understanding having intensely worked with survey answers.
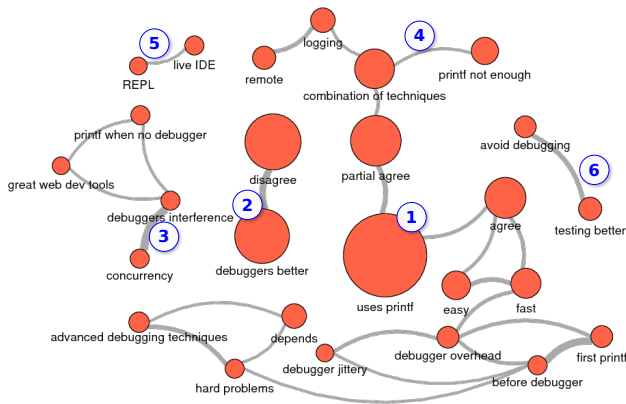
Figure 4: Intuitive visualization of the tag network in *SQ13*.

self-explanatory. 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 the two main strongly connected subgraphs ① and ② in Figure 4 suggest, there was a strong dichotomy between survey respondents: Many enthusiastically agreed with our statement ("Totally agree!", *SP13, SP23*), while others rejected it, stating that "[p]eople saying that never learned how to use a debugger" (*SR54*). Developers mostly seemed to agree that IDE debuggers are methodologically superior to print statements, explaining the strong link ②. Independently, reasons for resorting back to printf are when no debugger is available or when the presence of the debugger interferes with the program execution order ③. Many of the respondents who agreed with the statement also saw drawbacks of printf debugging, like *SP10*: "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 ④, as a combination of techniques is required. The 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 graphical user interface (GUI) can get in the way of working (fast). Instead of printf debugging, developers seemed to prefer a live IDE with a console that has a read-eval-loop (REPL) ⑤. Summarizing this discussion, *SP75* 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." Few developers also tried to avoid debugging by testing better ⑥.

4 IDE FIELD STUDY

In this section, we describe our field study with WATCHDOG 2.0.

4.1 Study Methods

Data Collection. To investigate the debugging habits of developers in the IDE, we extended the WATCHDOG infrastructure originally proposed by Beller et al. [57–59] to also track developers' debugging

behavior for *RQ2*, resulting in WATCHDOG 2.0 for both ECLIPSE and INTELLIJ. Beller et al. used WATCHDOG as a research vehicle to verify common expectations and beliefs about testing [57, 58]. From December 8th, 2015, to June 15th, 2016, we implemented functionality to track almost all debugging-relevant developer actions in the IDE in 211 commits.[4] 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 a new, orthogonal concept, singular events, that unlike intervals have no end date. Such events track when developers add, change or remove breakpoints, for example. An *IDE session* is an uninterrupted sequence of WATCHDOG intervals in which the developer does not close the IDE or suspend the computer. A *debugging session* is an IDE session, in which the developer used the debugger at least once.

Analysis Methods. To analyze the data collected with WATCHDOG 2.0, we created a data processing pipeline.[5] The pipeline, which comprises 4,000 lines of code, extracts the data from WATCHDOG's MONGODB database and loads it into R for further analysis. The analysis methods we used for some of these research questions require some more explanation detailed below.

For *RQ2.1* and *RQ2.2*, we assessed activity measured via WATCHDOG intervals. For *RQ2.4*, we assessed the intervals that occur before a debugging session is started. We chose a search range of 16 seconds before, matching the interval inactivity timeout of 16 seconds in WATCHDOG. This means that activity-based intervals like reading or typing intervals are automatically closed after this period of inactivity to account for e.g. coffee breaks. A timeout length of around 15 seconds is standard in IDE-based observational plugins [42, 58].

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.

Subject Recruitment. To attract participants to our field study, we relied on WATCHDOG's recruitment processes [58]. Users could join or leave the field study at any time.

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. Of the 458 users, 21% come from China, 12% from India, 12% from the US, 5% from Brazil, 4% from Germany, and the remaining 46% from 65 other countries. Users could opt to share their industrial programming experience, and 186 (41%) did: 68% had up to two years of programming experience, 16% between three to six years, and 16% seven years and more. Nine users were running MacOS X (5%), 14 Linux (6%), 162 Windows (89%), and 272 chose not to answer. Our study includes a heterogeneous mix of private, open-source, and commercial projects. Due to a privacy agreement to protect our users, we cannot disclose further details.

The median study participation was 6 days (mean: 13 days), the maximum the full 66 days. 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

---

[4] <identifying link removed>
[5] Open-Sourced here: <identifying link removed>

in which the IDE was open, which amounts to 10.3 observed developer years based on the 2015 average working hours for OECD countries [60]. We also collected 54,738 debugging events from 192 users, 218 projects and 723 IDE sessions. Only 48 users ($top_{10\%}$) are responsible for 90% of the sessions. In total, we recorded both at least one debug interval and one event for 108 users.

4.2 Results

In this section, we describe key results of our WATCHDOG 2.0 observational field study for RQ2.

RQ2.1: How prevalent and frequent is IDE debugging?

WR1: The majority of developers does not use the IDE-provided debugging infrastructure. Table 1 presents the number of occurrences of the different event types. 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. Of these, 108 study subjects (23.6%) have used the debugger and at least one of its features (transferred both intervals and events). In $top_{10\%}$, every user had at least one debugging session (100% debugger use). No debugger use is therefore likely a result of little transferred data. However, it is not contradictory to use the debugger and not transfer any of the events listed in Table 1, since the debugger provides several other benefits like hot-swapping of code. In total, we observed a debugger run in 9% of all 723 IDE sessions. In the onward analyses, we only take into account data from users who used the debugger.

WR2: About 20% *of the developers are responsible for over* 80% *of the debugging intervals in our sample.* For *RQ2.1* 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 data collection, 21% launched their debugger more than 20 times.

RQ2.2: How much time is spent in IDE debugging?

WR3: Debugging consumes, on average, less than 14% *of the active in-IDE development time.* For *RQ2.2*, 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 (0.1%), 721.5 hours of debugging intervals (4.0%), 2,568.8 hours of reading (14.1%), and 1,228.6 hours of typing (6.8%). These intervals are the main contributors of how developers spend their time in the IDE, included in the 'IDE active' intervals (28.9%). Next, we analyzed the duration and percentages on a per user basis. For the users with at least one debug interval, Table 2 shows the descriptive statistics of the interval duration and percentages. From the results in Table 2 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%.

WR4: Most debugging sessions consume less than 10 minutes. Furthermore, about half of the debugging sessions take at most 40 seconds, while about 12% of them last more than 10 minutes.

RQ2.3: Which IDE debugger features do developers use?

WR5: Line breakpoints are used most and by most developers, other breakpoint types are used less and by fewer developers. The results in Table 1 show 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.

WR6: Breakpoint options are not used by most WatchDog 2.0 *users; the most frequently used option is changing their enablement.* When considering how breakpoints evolve over their lifetime, the breakpoint change type frequencies in Table 1 (second column) 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, events related to specifying a hit count on the breakpoint have not been recorded during the collection period.

WR7: Setting breakpoints and stepping through code is done most, other debugging features are far less used. Table 1 shows that most of the recorded debugging events are related to the creation (4,544), removal (4,362) or adjustment 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.

RQ2.4: What is the relation between testing and debugging?

WR8: Most debugging sessions start after reading or changing the code, not after running tests. Regarding *RQ2.4*, 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 (0.5%) for running unit tests, 119 (1.2%) for other debug intervals, 4,991 (51.9%) for reading and 1,802 (18.7%) for typing intervals. About 70% of the debugging sessions start after reading or writing code, only 0.5% of them start after a failing or passing test run.

WR9: Developers who spend more time executing tests are likely to proportionally debug more. Next, we 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 duration spans.

WR10: Developers who read or modify test classes longer are not likely to debug less. To complete *RQ2.4*, we studied the relation between the amount of time the user spends inside test classes (i.e.,

Table 1: Frequencies of breakpoint types, modifications, and WATCHDOG 2.0 debugging events.

Breakpoint type	Frequency	Breakpoint modification	Frequency	Event type	Frequency	Event type	Frequency
Class prepare	99	Change condition	3	Add breakpoint	4,544	(continued)	
Exception	37	Disable condition	1	Change breakpoint	247	Resume client	8,292
Field	78	Enable condition	19	Remove breakpoint	4,362	Suspend by breakpoint	13,276
Line	4,229	Disable	180	Define watch	343	Suspend by client	16
Method	77	Enable	40	Evaluate expression	101	Step into	3,480
Undefined	24	Change suspend policy	4	Inspect variable	179	Step over	19,543
				Modify variable value	4	Step out	351
	Σ 4,544		Σ 247	(continuing ...)			Σ 54,738

Table 2: Descriptive usage statistics for key interval types (relative to total observed time).

Variable	Unit	Min	25%	Median	Mean	75%	Max	Log-Histogram
Debugging	Hours (%)	0.00 (0.0%)	0.03 (0.1%)	0.30 (0.5%)	5.47 (2.5%)	1.42 (2.4%)	333.70 (30.8%)	
Reading	Hours (%)	0.00 (0.0%)	0.14 (1.7%)	0.60 (3.2%)	5.70 (4.9%)	2.07 (5.7%)	591.10 (52.7%)	
Typing	Hours (%)	0.00 (0.0%)	0.21 (1.5%)	1.01 (3.6%)	2.95 (4.8%)	2.78 (6.9%)	63.87 (28.3%)	
Running JUNIT tests	Hours (%)	0.00 (0.0%)	0.00 (0.0%)	0.01 (0.0%)	0.68 (0.2%)	0.56 (0.2%)	9.19 (2.1%)	

either reads or modifies tests) 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 correlation ($\rho = 0.23$) when focusing on the 84 users with both at least one debug interval and one opened test class.

RQ2.5: How are file length and debugging effort related?

WR11: Smaller classes are debugged more than larger classes. Here we examined whether there is a correlation between the file size of a class (in source lines of code [61]), and the number of times the developer visits it in the source code editor in a debugging session. 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 apparent correlation ($\rho = 0.19$). For *RQ2.5*, we aggregated 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 4.8% (median: 1.7%) of all project classes we observed in WATCHDOG intervals were ever debugged. The value ranges from 0.2% to 100%, where the 100%-cases possibly stem from small projects with only one or two classes. For 2), the results range from 0.8% to 100% with an average of 14.5% (median: 4.5%). 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.

RQ2.6: Do developers often step over the point of interest?

WR12: Developers might step over the point of interest and have to start over again in 5% of debugging sessions. To answer *RQ2.6*, 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 3.2). For the 58 users that have

entered their experience and generated at least one debug interval, this resulted in a weak correlation ($\rho = 0.38$), i.e. more experienced developers are more likely to spend 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 the problem is by identifying possible cases of stepping over the point of interest. "Stepping over" means that the developer steps one time too far and has to start debugging all over again. Reasons for this include pressing the proceed key too fast or realizing too late that the actually interesting location was in a past step. To model this with our interval and event concept, we look for a set of debug intervals that satisfy the following 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. We then created subsets of these debug intervals by imposing a maximum time t_{max} between two consecutive debug intervals. Figure 5 shows the possible cases of stepping over the point of interest for the subsets with $t_{max} < 15$ minutes.
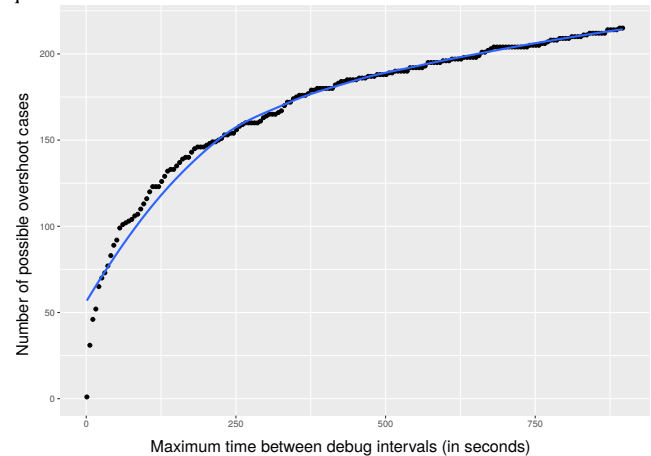

Figure 5: Possible cases of stepping over the point of interest per maximum time period between consecutive debug intervals.

Table 3: 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 Project Leaders		Switzerland, India	Eclipse Development
E2	Professor	> 20 years	Greece	Software Engineering
E3	Debugger Developer	18 years	Russia	IDE Development

The trend line in Figure 5 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.

5 INTERVIEWS

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

5.1 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 ran the combined observations from survey, objective IDE measurements, and anecdotal interview insights across two 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 67 minutes. In one case (*E3*), we performed the interview asynchronously via email. Subsequently, we transcribed the interviews and extracted insightful quotes.

Study Subjects. Table 3 gives an overview of our nine interviewees. We sampled the 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. We chose the experts based on their industrial and academic position in the debugging field.

5.2 Results

This section juxtaposes survey (*RQ1*) and IDE study (*RQ2*) results and discusses them with the qualitative insights from *RQ3*.

Use of the IDE Debugger. In *WR1*, 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 *SR1,* in which 80% of respondents claimed to use it. Moreover, no single user spent more than 30% of his development time debugging. 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 in *SR2*. 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 observation 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. Beller et al. observed a similar phenomenon with developers claiming to spend more time on testing than they really were [57]. As a consequence, we emphasize their 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-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. *SR3* and *SR4* 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 the debugger's other more advanced functions. *WR5* to *WR7* 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 who use the debugger have defined a watched expression. 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 the 16th, 17th, or 18th time?)" and that once one knows the condition, one almost automatically understands the problem. Then, there is no need for the conditional breakpoint anymore. Moreover, "the types of problems where you need a conditional breakpoint happen very rarely" (*I2*). For example, when we presented the breakpoint export feature of ECLIPSE to *I2*, he replied "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 an internal thought process not usually shared and that breakpoints are "like a one-shot. Ideally I wouldn't like them to be, but I just set them anew" (*I4*).

2) Debuggers are 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 mentor ships." 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 (*SR6*), 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 [62].

Time Effort for Debugging. Our study results *WR3* to *WR4* 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 (*WR3*) we found to consume significantly less than the $30 - -90\%$ for testing and debugging reported by Beizer [63] and 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 might be that developers (and humans in general) have a tendency to overestimate the duration of unpleasant tasks, as previously observed with testing [57]. Another is 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. A common intuition in Software Engineering is that "small is better," since it is easier to manage and understand, see for example the recommendations to micro services, small commits, or short files. Contrary to this claim, we found that short classes need considerably more debugging (*WR11*) and that the longer amount of time developers spend in larger classes does not nearly compensate for it. Our interviewees agreed in unison that the hardest problems to debug are ones where interfaces or transactions between components are involved. Interfaces are typically short since they contain little logic, but represent a common source of integration problems and thus, the answers suggest, debugging effort. Then, longer classes are likely to have increased locality of features, which makes them often easier to understand [64] and thus probably also easier to troubleshoot. We need more research on this unexpected interaction between file length and debugging probability. Future studies could try to exploit the finding to recommend optimal system designs as a compromise between modularity and the ability to debug them.

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 (*SR5, SR7*). However, there is mixed evidence on this in *RQ2*, as shown by *WR8, WR9* and *WR10*. 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 interesting 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 (*WR12*). This indicates that there is a limited, but existent gap in current debuggers process that might be filled by *back-in-time debuggers* [34]. Back-in-time debuggers 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, *WR12* 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 only found to be "helpful in a limited way" (*I3*). Currently, mainstream IDEs do not support back-in-time debugging.

Improvements in IDE Debuggers. 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 preserving all existing functionality. 2) Create tools that capture the holistic debugging process better. Elaborating 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 [65], to have a live REPL, a feature the IDE XCODE introduced [66].

To improve the design of existing IDE debuggers with findings from our study, 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 had sporadic evidence on how some individual developers use their debugger, they were unaware of the debugging behavior of a large population and the usage detail our study could provide. They started or updated six feature requests for the debugger based on our study, commencing work on bugs that had been dormant since 2004.[6] In the following, we focus on two already implemented features that are scheduled to roll out as part of Eclipse release 4.7.

In our field study and interviews, we identified left-over breakpoints as a recurrent annoyance, which developers have to remove manually, with *I1* saying that suspending on old breakpoints unexpectedly interrupts his flow and that "every so often, once a week or so, I just delete all of them." After making the Eclipse leads aware of this problem, they implemented age deprecation for breakpoints. It lets developers remove old breakpoints with one click. Although often referred to as a "dirty hack" (since it interferes with and pollutes production code), our study found that printf debugging also

---

[6] The umbrella bug 498469 describes and links to the sub change requests, see <identifying link removed>.

provides an advantage over the debugger's watch view in that it preserves the history of past logs (for example, of memory entities in the watch view). Conversely, developers cannot enrich third-party libraries for which no source code is available with printf statements, but they can place debugger breakpoints in e.g. their Java byte code. To keep a history of logs when using the debugger, before our study, Eclipse and IntelliJ users had to set up an artificial construction of placing a conditional breakpoint that would print the information and always return false, thus never suspend. This hack of a "conditional breakpoint that is not conditional" (Bugtracker description) required intimate familiarity with the idiosyncrasies of the debugger and had bad performance, since code embedded in conditional breakpoints runs via the Java Debugging Infrastructure, which adds unnecessary overhead for a simple printout. By offering the new breakpoint type "tracepoint," developers can now conveniently produce fast logs of debug traces. The Eclipse project implemented this simplified solution in Bug 71020, which had been in hibernation since 2004 and on which work commenced after our discussion.

6 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. However, such outliers are expected in an observational study of several months [58]. 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.

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 is 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. Due to the fact that WATCHDOG gathers data automatically, it is harder for potentially evil-minded users to fabricate data than in surveys. Moreover, that the majority of data comes from a relatively small "power user" population (48 developers in our case, $top_{10\%}$ in *WR1*) is both normal in service use, for example on Twitter [67], and other observational studies [42, 68]. Discrepancies between some survey answers and the objective IDE observations have previously

been observed in other studies [58].
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.

7 CONCLUSION

We set out to obtain a first cross-validated understanding of developers' debugging knowledge and contrasted it with their real-world IDE debugging behavior.

We found strong dichotomies in developers' opinions, knowledge, and behavior: Many believe modern debuggers to be superior to printf debugging, yet still employ it for many good reasons. IDE observations confirmed this finding, as only a third of developers ever invoked the debugger. We found that debugging is a technique defined by necessities: It is a relatively fast-paced and short-lived activity that is by nature so complicated that the tools around it should be as simple as possible. Consequently, developers use only basic features and seldom resort to more advanced breakpoint types or debugging techniques. Developers spend surprisingly little time in the debugger; only 13% of their total development time on average, in stark contrast to previous findings claiming more than 50%. As developers become more experienced, they seem to use the debugger slightly more, possibly because they educated themselves on its advanced affordances over printf debugging. We also found that having more tests in the code generally does not reduce the debugging burden, possibly because test code adds to the overall code that needs debugging.

In general, developers' theoretical knowledge and practical use of specialized debugging features are relatively shallow, just the amount that is seemingly sufficient for their debugging problems. Most developers said debugging was self-taught and not part of their curriculum. We believe that more educators can include practical, hands-on teaching, start in first year courses. Astonishingly, although bugs are inevitably linked with software and students learn programming in their introductory courses, they are only taught to properly debug much later, if ever.

Adding to this lack of debugging education, even experienced developers admitted that debuggers are not easy to use. Apart from the wish for back-in-time debuggers, developers never expressed the wish for more debugging features. Instead of introducing ever more esoteric features, we therefore call to make using the already existing debugger features easier to use and more accessible. With the help of three ECLIPSE project leads, we identified several areas of improvement in the ECLIPSE debugger, leading to new simplified debugging features. One example is the introduction of a new breakpoint type in Eclipse that combines the advantages of debugger-instrumentation with the flexibility of printf debugging. Other IDE and debugger

creators could follow this example and use our findings to further improve their debuggers.

ACKNOWLEDGMENTS

REFERENCES

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

[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] B. W. Kernighan and P. J. Plauger, "The elements of programming style," *The elements of programming style, by Kernighan, Brian W.; Plauger, PJ New York: McGraw-Hill, c1978.*, 1978.

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

[5] 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.

[6] P. Oman, C. Cook, and M. Nanja, "Effects of programming experience in debugging semantic errors," *Journal of Systems and Software*, vol. 9, no. 3, pp. 197–âĂŞ207, 1989.

[7] 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.

[8] 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.

[9] 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. ACM, 2010, pp. 61–72.

[10] 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.

[11] 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.

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

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

[14] E. Adams and S. S. Muchnick, "dbxtool: A window-based symbolic debugger for Sun workstations," *Software: Practice and Experience*, vol. 16, no. 7, pp. 653–669, Jul. 1986.

[15] N. Matloff and P. J. Salzman, *The Art of Debugging with GDB, DDD, and Eclipse*. San Francisco: No Starch Press, 2008.

[16] B. Beander, "VAX DEBUG: An interactive, symbolic, multilingual debugger," in *Proceedings of the Software Engineering Symposium on High-Level Debugging*, M. Johnson, Ed. ACM SIGSOFT/SIGPLAN, Mar. 1983, pp. 173–179.

[17] B. Tuthill and K. J. Dunlap, "Debugging with dbx," in *UNIX Programmer's Supplementary Documents, Volume 1*. Berkeley, California 94720: Computer Systems Research Group, Department of Electrical Engineering and Computer Science, University of California, Apr. 1986, 4.3 Berkeley Software Distribution.

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

[19] A. Ko and B. Myers, "Debugging reinvented," in *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*. IEEE, 2008, pp. 301–310.

[20] D. J. Gilmore, "Models of debugging," *Acta psychologica*, vol. 78, no. 1, pp. 151–172, 1991.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] C. Zhang, J. Yang, D. Yan, S. Yang, and Y. Chen, "Automated breakpoint generation for debugging," *Journal of Software*, vol. 8, no. 3, 2013.

[26] 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. ACM, 2012, pp. 309–319.

[27] 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.

[28] 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.

[29] M. Perscheid and R. Hirschfeld, "Follow the path: Debugging tools for test-driven fault navigation," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*. IEEE, 2014, pp. 446–449.

[30] 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.

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

[32] G. C. Murphy, M. Kersten, and L. Findlater, "How are Java software developers using the Elipse IDE?" *IEEE software*, vol. 23, no. 4, pp. 76–83, 2006.

[33] C. Parnin and S. Rugaber, "Resumption strategies for interrupted programming tasks," *Software Quality Journal*, vol. 19, no. 1, pp. 5–34, 2011.

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

[35] M. Perscheid, B. Siegmund, M. Taeumel, and R. Hirschfeld, "Studying the advancement in debugging practice of professional software developers," *Software Quality Journal*, vol. 25, no. 1, pp. 83–110, 2016.

[36] L. Layman, M. Diep, M. Nagappan, J. Singer, R. Deline, and G. Venolia, "Debugging revisited: Toward understanding the debugging needs of contemporary software developers," in *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*. IEEE, 2013, pp. 383–392.

[37] 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)*. IEEE, 2015, pp. 11–20.

[38] 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.

[39] M. Böhme, E. O. Soremekun, S. Chattopadhyay, E. Ugherughe, and A. Zeller, "How developers debug software the dbgbench dataset: Poster," in *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE Companion)*. IEEE, 2017, pp. 244–246.

[40] ——, "Where is the bug and how is it fixed? an experiment with practitioners," in *Proceedings of the 11th Joint Meeting of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. IEEE, 2017, pp. ??–??.

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

[42] S. Amann, S. Proksch, S. Nadi, and M. Mezini, "A study of visual studio usage in practice," in *23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 124–134.

[43] L. Hattori and M. Lanza, "Syde: a tool for collaborative software development," in *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2010, pp. 235–238.

[44] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig, "A comparative study of manual and automated refactorings," in *Proceedings of the 27th European Conference on Object-Oriented Programming*, 2013.

[45] P. M. Johnson, "Searching under the streetlight for useful software analytics," *IEEE software*, vol. 30, no. 4, pp. 57–63, 2013.

[46] B. W. Kernighan, *UNIX for Beginners*. Bell Laboratories Murray Hill, NJ, 1978.

[47] D. Spinellis, "Debuggers and logging frameworks," *IEEE Software*, vol. 23, no. 3, pp. 98–99, May/June 2006.

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

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

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

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

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

[53] C. S. Horstmann and G. Cornell, *Core Java 2: Volume I, Fundamentals*. Pearson Education, 2002.

[54] J. Ressia, A. Bergel, and O. Nierstrasz, "Object-centric debugging," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 485–495.

[55] A. Chiş, T. Gîrba, and O. Nierstrasz, "The moldable debugger: A framework for developing domain-specific debuggers," in *International Conference on Software Language Engineering*. Springer, 2014, pp. 102–121.

[56] 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

[57] 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. ACM, 2015, pp. 179–190.

[58] 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.

[59] 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. ACM, 2016, pp. 53–56.

[60] Organisation 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].

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

[62] D. H. O'Dell, "The debugging mind-set," *Communications of the ACM*, vol. 60, no. 6, pp. 40–45, 2017.

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

[64] "Remodularizing Java programs for improved locality of feature implementations in source code," *Science of Computer Programming*, vol. 77, no. 3, pp. 131–151, 2012.

[65] C. Systems, "Chronon, a DVR for Java," http://chronosystems.com/, 2015, [Online; accessed 24 August 2016].

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

[67] G. M. Chen, "Tweet this: A uses and gratifications perspective on how active twitter use gratifies a need to connect with others," *Computers in Human Behavior*, vol. 27, no. 2, pp. 755–762, 2011.

[68] R. Minelli, A. Mocci, and M. Lanza, "I know what you did last summer: an investigation of how developers spend their time," in *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*. IEEE Press, 2015, pp. 25–35.