2017 IEEE International Conference on Software Quality, Reliability and Security

How Do Developers Toggle Breakpoints?
– Observational Studies –

Fabio Petrillo, Hyan Mandian, Aiko Yamashita, Foutse Khomh, Yann-Gaël Guéhéneuc
Polytechnique Montréal, Canada
UniRitter, Brazil
Oslo and Akershus University College of Applied Sciences, Norway
E-Mails: fabio@petrillo.com, hyanmandian@hotmail.com,
aiko.yamashita@hioa.no,foutse.khomh@polymtl.ca,yann-gael.gueheneuc@polymtl.ca

Abstract—**One of the most important tasks in software maintenance is debugging. Developers perform debugging to fix faults and implement new features. Usually they use interactive development environments to perform their debugging sessions. To start an interactive debugging session, developers must set breakpoints. Choosing where to set breakpoints is a non-trivial task, yet few studies have investigated how developers set breakpoints during interactive debugging sessions. To understand how developers set breakpoints, we analysed more than 10 hours of 45 video-recorded debugging sessions, where a total of 307 breakpoints were set. We used the videos from two independent studies involving three software systems. We could observe that: (1) considerable time is spent by developers until they are able to set the first breakpoint; (2) when developers toggle breakpoints carefully, they complete tasks faster than developers who set (potential useless) breakpoints quickly; and (3) different developers set breakpoints in similar locations while working (independently) on the same tasks or different tasks. We discuss some implications of our observations for debugging activities.**

Index Terms—**Debugging, breakpoints, interactive debugging, empirical software engineering.**

I. Introduction

Debugging is an important activity during software development, maintenance, and evolution [1]. Latoza *et al.* [2] showed that developers spend at least 30% of their development time on debugging activities. Thus, supportive features for this process could increase developer's productivity. Furthermore, in a recent study about change impact analysis, Jiang *et al.* [3] observed that 40% of the developers go to debug mode directly after they have performed a change, and 63% of the participants used dynamic approaches (e.g., debugging) to perform impact analysis over their changes.

During debugging sessions, developers use debuggers to detect, locate, and correct faults in software systems. Interactive debugging is one particular debugging process in which developers use interactive tools to investigate the execution of a system. Modern interactive debuggers are often integrated in development environments (IDEs), such as Eclipse, Netbeans, Intellij IDEA, and Visual Studio, enabling navigating through the code and setting breakpoints.

In general, developers initiate an interactive debugging session by setting a breakpoint. Setting a breakpoint is one of the most frequently used features of IDEs [4]. To decide where to set a breakpoint, developers must use their observations, recall their experiences with similar tasks, formulate hypotheses about the task, and exercise a judgment [5]. Tiarks and Röhms [6] observed that developers have difficulties in finding locations for setting the breakpoints, suggesting that this is a difficult activity, and that supporting developers to set appropriate breakpoints could reduce debugging effort.

Yet, to the best of our knowledge, no study has been performed on how developers set breakpoints. In particular, no study has involved participants correcting the same set of bugs (i.e., performing the same debugging tasks). A study with such characteristics can provide useful insights to researchers and tool developers on how to provide appropriate support during debugging activities in general, and particularly when deciding where to set breakpoints. It could also help build a grounded theory on the setting of breakpoints by developers to improve debuggers and other tool support.

To fill this gap in the literature, we conducted two observational studies with the aim to understand how developers set breakpoints. We collected and analysed more than 10 hours of videos from 45 interactive debugging sessions. We collected the videos from two completely independent studies on 3 different software systems. Our studies involved 28 different, independent developers, who set a total of 307 breakpoints to fault locations tasks and fault-fixing tasks.

Our studies aimed at answering the following four research questions about breakpoints and breakpoints setting:

RQ1: **What is the effort (time) for setting the first breakpoint in relation to the total effort for a debugging task?** We observe that setting the first breakpoint takes on average 25% of the total duration of the debugging tasks.

RQ2: **Is there a correlation between time of first breakpoint and tasks elapsed time?** We observe a clear, inverse correlation between task elapsed-time and time of first breakpoint ($\rho = -0.47$).

RQ3: **Are there consistent, common debugging trends wrt. the types of statements among developers?** We observe that 50% of the breakpoints are set on method calls.

RQ4: **Are there consistent, common debugging trends wrt. setting breakpoints on the same line, method, or class, among developers?** A surprising result is that different developers, for different tasks, set breakpoints at the same locations.

978-1-5386-0592-9/17 $31.00 © 2017 IEEE
DOI 10.1109/QRS.2017.39

The remainder of the paper is organized as follows. Section II provides some background about debugging. Section III presents the design of our study. Section IV shows and discusses the answer to our research questions. Section V discusses threats to the validity of our study. Section VI summarizes related work. Finally, Section VII concludes the paper and outlines future work.

II. Background

This section provides background information about the debugging activity and setting breakpoints. In the following, we use **defect** as unintended behaviour of the program, i.e., when the program does something that it should not, and **fault** as the incorrect statements in source code causing a defect. The purpose of debugging is to locate and correct faults.

A. Debugging

The IEEE Standard Glossary of Software Engineering Terminology defines debugging as the act of detecting, locating, and correcting faults in a computer program. Debugging techniques include the use of breakpoints, desk checking, dumps, inspection, reversible execution, single-step operations, and traces.

Araki *et al.* [7] describe *debugging* as a process where developers make hypotheses about the root-cause of a problem or defect and verify these hypotheses by examining different parts of the source code of the program.

Interactive debugging consists of using a tool, i.e., *a debugger* to detect, locate, and correct a fault in a program. It is a process also known as *program animation*, *stepping*, or *following execution* [8]. Developers often refer to this process as simply *debugging*, because several IDEs provide debuggers to support *debugging*. However, it must be noted that while *debugging* is the process of finding faults, *interactive debugging* is one particular debugging approach in which developers use interactive tools. Expressions such as *interactive debugging*, *stepping* and *debugging* are used interchangeably, and there is not yet a consensus on what is the best name for this process.

B. Breakpoints and Supporting Mechanisms

Generally, breakpoints allow to pause the execution of a program and to examine the call stack and variable values when the control flow reaches the locations of the breakpoints. Thus, a breakpoint indicates the location (line) in the source code of a program where a pause will occur during its execution. Depending on the programming language, its run-time environment (in particular the capabilities of its virtual machines if any), and the debuggers, different types of breakpoints may be available to developers. These types include static breakpoints [9], that pause the execution of a program, and dynamic breakpoints [10], that pause depending on some conditions or threads or numbers of hits. Other types of breakpoints include wathcpoints, that pause the execution when a variable being watched is read and–or written. IDEs offer the means to specify the different types of breakpoints depending on the programming languages and their run-time environment. Fig. 1-A and 1-B show examples of static and dynamic breakpoints in Eclipse. In the rest of this paper, we focus on static breakpoints because they are the most used of all types [5].

There are different mechanisms for setting a breakpoint within the code:

- *GUI:* Most IDEs or Browsers offer a visual way of adding a breakpoint, usually by clicking at the beginning of the line on which to set the breakpoint: *Chrome*[1], *Visual Studio*[2], *IntelliJ*[3], and *Xcode*[4].
- *Command line:* Some programming languages offer debugging tools on the command line, so an IDE is not necessary to debug the code: *JDB*[5], *PDB*[6], and *GDB*[7].
- *Code:* Some programming languages allow using syntactical elements to set breakpoints as they were 'annotations' in the code. This approach often only supports the setting of a breakpoint, and is necessary to use in conjunction with the command line or GUI. Some examples are: *Ruby debugger*[8], *Firefox*[9], and *Chrome*[10].

There is a set of features in a debugger that allow developers to control the flow of the execution within the breakpoints i.e., *Call Stack features*, which enable *continuing* or *stepping*.

A developer can opt for *continuing*, in which case the debugger will continue execution until the next breakpoint is reached or the program exits. Conversely, *stepping* allows the developer to run step by step the entire program flow. The definition of a step varies across programming languages and debuggers, but it generally includes invoking a method and executing a statement. While Stepping, a developer can navigate between *steps* using the following commands:

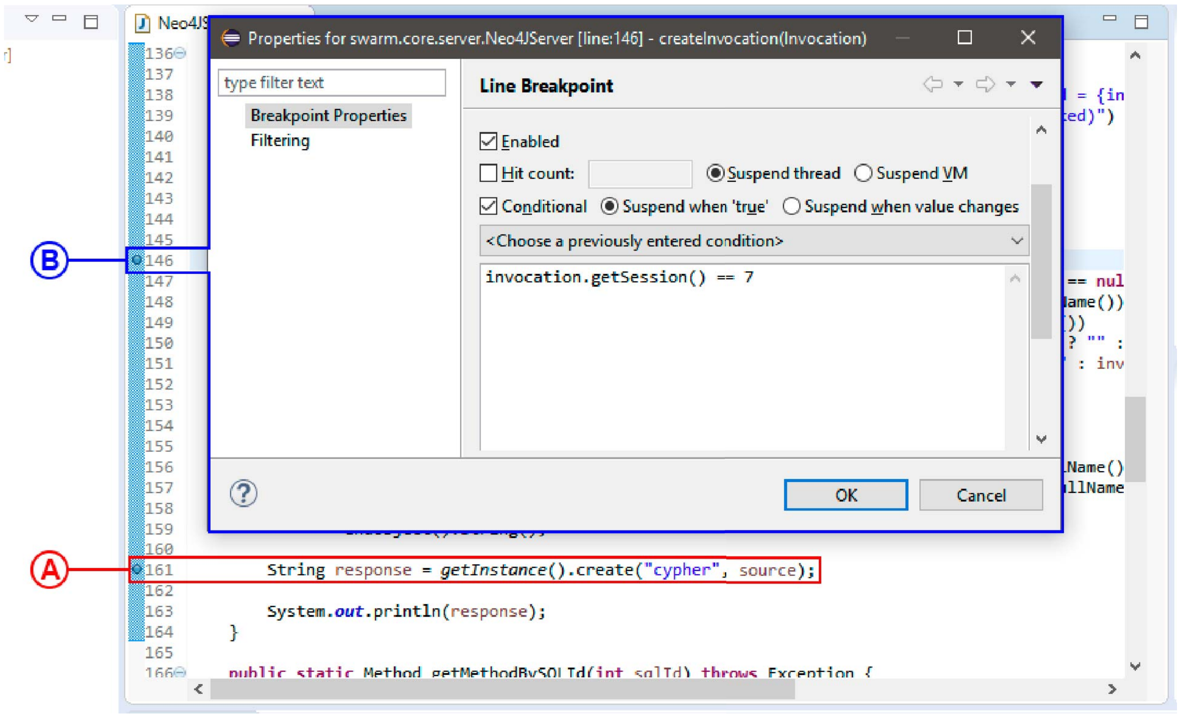
- *Step Over:* the debugger will step over a given line. If the line contains a function, then the function will be executed and the result returned without stepping through each of its lines.
- *Step Into:* the debugger will enter the function at the current line and continue stepping there line-by-line.
- *Step Out:* This action would take the debugger back to the line where the current function was called.

Finally, some debuggers allow debugging *remotely*, for example to perform hot-fixes or to test mobile applications and systems operating in remote configurations. Table I provides a summary on twelve popular development frameworks that offer support for debugging.

III. Study Design

Our analysis builds upon two independent sets of observations involving in total, 3 systems. Study 1, involves *JabRef* as

[1] https://developers.google.com/web/tools/chrome-devtools/javascript/add-breakpoints
[2] https://msdn.microsoft.com/en-us/library/5557y8b4.aspx
[3] https://www.jetbrains.com/help/idea/2016.3/debugger-basics.html
[4] http://jeffreysambells.com/2014/01/14/using-breakpoints-in-xcode
[5] http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html
[6] https://docs.python.org/2/library/pdb.html
[7] ftp://ftp.gnu.org/oldgnu/Manuals/gdb5.1.1/html_node/gdb_37.html
[8] https://github.com/cldwalker/debugger
[9] https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Statements/debugger
[10] https://developers.google.com/web/tools/chrome-devtools/javascript/add-breakpoints



Properties for swarm.core.server.Neo4JServer [line:146] - createInvocation(Invocation)

Line Breakpoint

[✓] Enabled

[ ] Hit count: [✓ Suspend thread] Suspend VM

[✓] Conditional [✓ Suspend when 'true'] Suspend when value changes

<Choose a previously entered condition>

```
invocation.getSession() == 7
```

```
        String response = getInstance().create("cypher", source);

        System.out.println(response);
    }

    public static Method getMethodBySQLId(int sqlId) throws Exception {
```

Fig. 1: Setting a static breakpoint (A) and a conditional breakpoint (B) using Eclipse IDE

TABLE I: Popular IDEs and their debugging functionality

IDE Browser/CLI	Static Breakpoint	Conditional Breakpoint	Call Stack	Remote
Google Chrome	Yes [11]	Yes [11]	Yes [12]	Yes [13]
Firefox	Yes [14]	Yes [14]	Yes [14]	Yes [15]
Eclipse	Yes [16]	Yes [16]	Yes [16]	Yes [17]
IntelliJ	Yes [18]	Yes [18]	Yes [18]	Yes [19]
NetBeans	Yes [20]	Yes [21]	Yes [22]	Yes
Visual Studio	Yes [23]	Yes [23]	Yes [23]	Yes [24]
PDB	Yes [25]	Yes [25]	Yes [25]	No [26]
Xcode	Yes [27]	Yes [27]	Yes [27]	No
GDB	Yes [28]	Yes [28]	Yes [28]	Yes [28]
JDB	Yes [29]	No	Yes [29]	Yes [30]
Android Studio	Yes [31]	Yes[31]	Yes [31]	Yes [32]
Cloud9	Yes [33]	Yes [33]	Yes [33]	Yes [34]

the subject system, and Study 2 involves *PdfSam* and *Raptor* systems. We analysed 45 video-recorded debugging sessions, available from our own collected videos (Study 1) and from an empirical study performed by Jiang *et al.* [3] (Study 2).

A. Study 1: Observational Study on JabRef

1) Subject program: We selected JabRef[11] version 3.2 as subject program. JabRef is a bibliography reference manager developed in Java. One advantage of this system is that since it is an Open Source project, defect reports are publicly available via its issue tracking system. Also, its domain is easy to understand (thus reducing the context *learning effect*), and it is composed of relatively independent packages and classes i.e., high cohesion, low coupling (thus reducing the potential commingle effect of *low code quality*).

2) Participants: We recruited eight male professional developers via an internet-based freelancer service[12]. Two participants are experts and three are intermediate in Java. Developers self-reported their expertise levels, which thus should be taken with caution. In addition, we recruited 12 volunteers among undergraduate and graduate students at Polytechnique Montréal to participate in our observational study. All participants stated that they are comfortable, and work regularly, with the debugging features of Eclipse.

A survey was applied before the study to collect all the participants' background information. The survey included questions about participants' self-assessment on their level of programming expertise.

3) Task Description: We selected five defects reported in the defect tracking system of JabRef, making sure that the task of fixing the defects would require developers to set breakpoints in different Java classes. We asked participants to find the locations of the faults described in issues 318, 667, 669, 993, and 1026. Table II summarises the faults using their titles in the tracking system.

4) Artifacts and Working Environment: We provided the participants with two documents: The first document was a tutorial [13] explaining how to install and configure the tools

[11]http://www.jabref.org/

[12]https://www.freelancer.com/

[13]http://swarmdebugging.org/publication

TABLE II: Summary of the faults considered in JabRef in Study 1

Issue	Summary
318	"Normalize to Bibtex name format"
667	"hash/pound sign causes URL link to fail"
669	"JabRef 3.1/3.2 writes bib file in a format that it will not read"
993	"Issues in BibTeX source opens save dialog and opens dialog *Problem* with parsing entry' multiple times"
1026	"Jabref removes comments inside the Bibtex code"

required for the study, and how to carry out a warm-up task. A video [14] was also presented for guiding the participants with the warm-up task.

The second document described the five faults and steps to reproduce them. We also provided participants with a video demonstrating step-by-step how to reproduce the five defects, to help them getting started.

We provided a pre-configured Eclipse workspace to the participants and asked them to install Java 8, Eclipse Mars 2 with [blind] plug-in to automatically collect breakpoint-related events. We required the participants to install and configure the Open Broadcaster Software[15] (OBS), an open-source system for live streaming and recording.

The Eclipse workspace contained two Java projects: a Tetris game for the warm-up task, and JabRef v3.2 for the study.

5) Study Procedure: After installing their environments, participants were asked to perform a warm-up task with a Tetris game. The task consisted of starting a debugging session, setting a breakpoint, and debugging the Tetris program to locate a given method. We used this task to confirm that the participants' environments were properly configured, and also to get the participants accustomed with the study setting. It was a trivial task that we used to filter the participants. All participants who participated in our study executed correctly the warm-up task[16].

After performing the warm-up task, each participant performed debugging to locate the faults. We established a maximum limit of one-hour per task, and informed the participants that the task would required about 20 minutes for each fault. This estimation was based on previous experiences with the tasks. After the participants were done each task, we administered an on-line post-experiment questionnaire to collect information about the study, asking if they found the fault, where was the fault, why the fault happened, if they were tired, and a general summary of their debugging experience.

6) Data Collection: All debugging data (breakpoints, stepping, method invocations) were automatically and transparently collected by the [blind] plug-in. In addition, we recorded the participant's screens during their debugging sessions by using OBS. During 8 days, we collected the following data:

[14]https://youtu.be/U1sBMpfL2jc

[15]https://obsproject.com

[16]We applied that warm-up task for 30 of freelancers, but only eight freelancers performed the task correctly.

- Video captures, one per participant, per task. The videos are essential to control the quality of each execution and to produce a reliable and reproducible chain of evidence for our results.
- The statements (including line and position in the source code) where the participants set breakpoints. We considered the following types of statements because they are representative of the main concepts in any programming languages:
  - *call*: method/function invokation;
  - *return*: return of values;
  - *assignment*: setting of values;
  - *if-statement*: conditional statement;
  - *while-loop*: loop, iteration.
- Summary of the results of the study, via an online questionnaire, including the following questions:
  - Did you locate the bug/issue?
  - Where is the bug/issue?
  - Why does the bug/issue happen?
  - Were you tired?
  - Describe your debugging experience.

Based on this data, we computed the following metrics, per participant and per task:

- Start Time (ST): the effective time when a developer starts a task. We analysed each video and we started to count when effectively the developer started a task (i.e., when she started the [blind] plug-in for example).
- Time of First Breakpoint (FB): the time when a developer set the first breakpoint.
- End time (T): the time when a developer finished a task.
- Elapsed End time (ET): $ET = T - ST$
- Elapsed Time First Breakpoint (EF): $EF = FB - ST$

We verified manually whether participants were successful or not at completing their task by analyzing the answers provided in the questionnaire and the videos (the location of the faults were available since all tasks were solved by JabRef's developers).

B. Study 2: Empirical Study on PdfSam and Raptor

The second study consisted of a re-analysis of 20 videos of debugging sessions available from a study conducted by Jiang *et al.* [3]. Jiang *et al.* presented an empirical study on change impact analysis by professional programmers. They conducted this study in two phases: In the first phase, they asked nine programmers to read two defect reports from two open-source systems and to fix the faults. The objective was to observe the programmers' behavior as they fixed the faults. They also analysed the developers' behavior in order to determine whether the developers used any tools for change impact analysis and, if not, whether they performed change impact analysis via manual inspection.

The two systems analysed in their study are "PDF Split and Merge" (PdfSam)[17] and Raptor[18]. They chose one defect

[17]http://www.pdfsam.org/

[18]https://code.google.com/p/raptor-chess-interface/

TABLE III: Elapsed time by task (average) - Study 1 (JabRef) and Study 2

Task	Average Times (min.)	Std Dev (min.)
318	44	64
667	28	29
669	22	25
993	25	25
1026	25	17
PdfSam	54	18
Raptor	59	13

report per system for their study. The systems were chosen due to their non-trivial size, and because the purpose/domain of the systems was clear and easy to understand. The choice of the defect reports followed the criteria that they were already solved and that they could be easily understood by developers who did not know the systems beforehand. Alongside each defect report, the participants were presented with information about the systems, their purpose, the main entry points, and instructions for replicating the defect.

C. Summary of Studies

Studies 1 and 2 have different approaches. The tasks in Study 1 were fault location tasks. Developers did not correct the faults. However, the tasks in Study 2 were fault correction tasks. In addition, Study 1 explored five different defects while Study 2 only analysed one defect per system. The collected data set provides a diversity of cases, and a rich, in-depth overview of how developers set breakpoints during different debugging activities. To support reproducibility by independent researchers, we plan to release all the data online[19], including: the anonymised videos, the logged activity sequences, and the results.

IV. Results

We analysed in total, 45 debugging sessions performed by 28 developers, comprising in total, 307 breakpoints and more than 10 hours of footage. In the following, we present the analyses performed to answer our research questions.

RQ1: What is the effort (time) for setting the first breakpoint in relation to the total effort for a debugging task?

We normalised the elapsed time between the start of a debugging session and the setting of the first breakpoint (EF) by dividing it by the total duration of the task (ET), to compare the performance of participants across tasks (1).

$$MFB = \frac{EF}{ET} \tag{1}$$

Table III shows the average effort (in minutes) for each task. We find in Study 1 that, in average participants spend 27% of the total task duration to set the first breakpoint (std. dev. 17%). In Study 2, it took in average 23% of the task time to participants to set the first breakpoint (std. dev. 17%).

We conclude that the effort for setting the first breakpoint takes near one quarter of the total effort of a single debugging session. This effort is important and hints that debugging time could be reduced by providing tool support for setting breakpoints.

RQ2: Is there a correlation between time of first breakpoint and tasks elapsed time?

For each session, we normalized the data using Eq. 1 and associated the ratios with their respective task elapsed times. Figure 2 combines the data from the 45 interactive debugging sessions where each point is a breakpoint by time. Analysing the first breakpoint data, we also found a clear correlation between task elapsed time and time of first breakpoint ($\rho = -0.47$), finding that task elapsed time has a correlation inversely proportional to the time of task's first breakpoint, following a correlation function:

$$f(x) = \frac{\alpha}{x^{\beta}} \tag{2}$$

where $\alpha = 12$ and $\beta = 0.44$.

We observe that when developers toggle breakpoints carefully, they complete tasks faster than developers who set (potential useless) breakpoints quickly.

The finding also corroborates previous results by Petrillo et *et al.* [35] on a larger set of tasks.

RQ3: Are there consistent, common debugging trends wrt. the types of statements among developers?

We classified the types of statements on which the participants set their breakpoints. We analysed each breakpoint to obtain Table IV for Study 1, which shows for example that 53% (111/207) of breakpoints are set on **call statements** while only 1% (3/207) are set on while-loop statements. For Study 2, Table V also shows similar trends: 43% (43/100) of breakpoints are set on **call statements** and only 4% (3/207) on while-loop statements. The only difference is on assignment statements, where Study 1 displays 17% whiles Study 2 displays 27%. After grouping *if-statement*, *return*, and *while-loop* into *control-flow* statements, we report that 29% of breakpoints are on control-flow statements.

Our results show that in both studies, 50% of the breakpoints were set on call statements while control-flow related statements were comparatively low, being the while-loop statement the least common (2-4%)

[19]http://[blind]

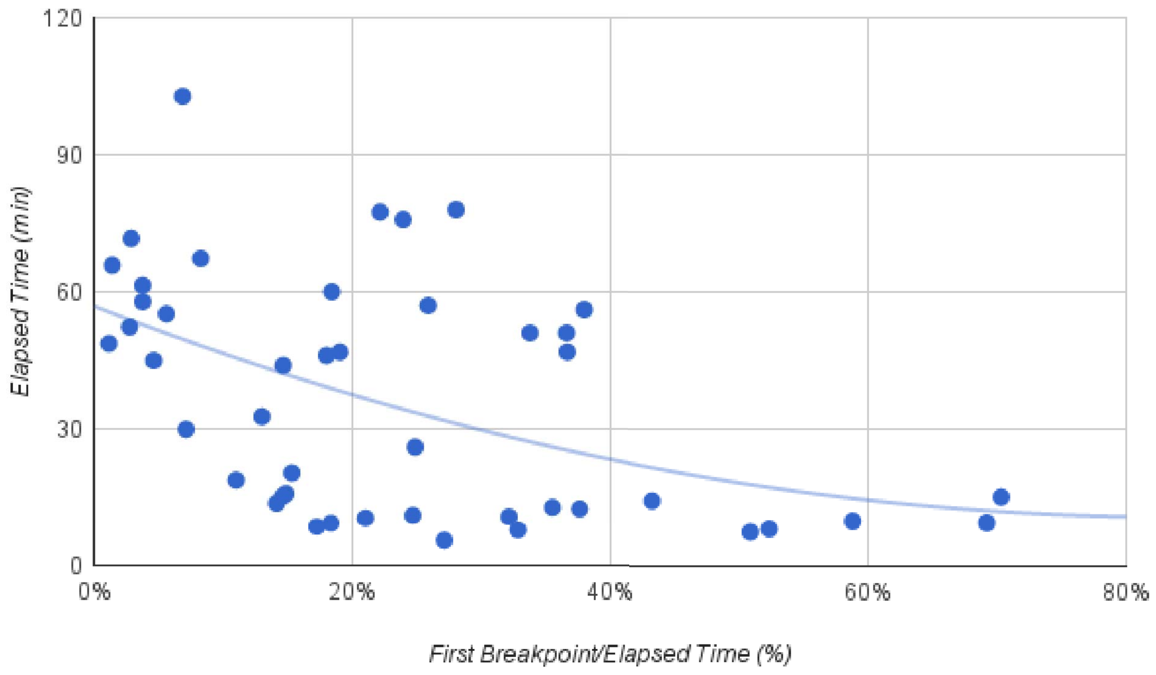


Fig. 2: Relation between time of first breakpoint and task elapsed time on the two studies

RQ4: Are there consistent, common debugging trends wrt. setting breakpoints on the same line, method, or class, among developers?

We investigate each breakpoint to assess whether there were breakpoints on the same line of code for different participants, performing the same tasks (i.e., resolving the same fault) by comparing the breakpoints on the same task and on different tasks. We sorted all the breakpoints from our data by the *Class* in which they were set and *line number* and we counted how many times a breakpoint was set on *exactly the same line of code* across participants. We report the results in Table VI for Study 1 and in Tables VII and VIII for Study 2.

In Study 1, we found 15 lines of code having two or more breakpoints on the same line of code for a same task by different developers. In Study 2, we observe breakpoints on exactly same line of code for eight lines of code in PdfSam project and six lines of code in Raptor. For example, in the Study 1, on the line of code 969 in the *Class BasePanel*:

TABLE IV: Study 1 - Breakpoints per type of statement

Statement	Number of Breakpoints	%
call	111	53
if-statement	39	19
assignment	36	17
return	18	10
while-loop	3	1

TABLE V: Study 2 - Breakpoints per type of statement

Statement	Number of Breakpoints	%
call	43	43
if-statement	22	22
assignment	27	27
return	4	4
while-loop	4	4

```
JabRefDesktop.openExternalViewer(metaData(),
   link.toString(), field);
```

Three different developers set three breakpoints on that line for task *0667*. Tables VI, VII and VIII report all recurrent breakpoints. These observations show that developers do not choose breakpoints lightly, as suggested before by [6]. The data suggests that there is an *underlying rationale* on that decision because different developers set breakpoints on exactly the same lines of code.

Surprisingly, analysing also the recurrent breakpoints independently of task, we produce Table IX to Study 1. Each line of Table IX shows a *Class*, the lines of code with a recurrent breakpoint separated by commas and the respective number of breakpoint occurrences in that line separated by commas[20]. For

[20]We arrange several lines of code in one single line because of paper's pages limitation.

TABLE VI: Study 1 - Breakpoints in the same line of code (JabRef) by task

Task	Class	Line of code	Breakpoints
0318	AuthorsFormatter	43	5
0318	AuthorsFormatter	131	3
0667	BasePanel	935	2
0667	BasePanel	969	3
0667	JabRefDesktop	430	2
0669	OpenDatabaseAction	268	2
0669	OpenDatabaseAction	433	4
0669	OpenDatabaseAction	451	4
0993	EntryEditor	717	2
0993	EntryEditor	720	2
0993	EntryEditor	723	2
0993	BibDatabase	187	2
0993	BibDatabase	456	2
1026	EntryEditor	1184	2
1026	BibtexParser	160	2

TABLE VII: Study 2 - Breakpoints in the same line of code (PdfSam)

Class	Line of code	Breakpoints
PdfReader	230	2
PdfReader	806	2
PdfReader	1923	2
ConsoleServicesFacade	89	2
ConsoleClient	81	2
PdfUtility	94	2
PdfUtility	96	2
PdfUtility	102	2

example, the Class *EntryEditor* had four breakpoints on the line of code *720*. Analysing Table IX, we found 135 lines of code having two or more breakpoints on the same line of code for different tasks by different developers. For example, five different developers set five breakpoints on that line on the line of code 969 in the *Class BasePanel* independently of task (in that case on 3 different tasks). This result suggest a potential opportunity to recommend those locations as candidates for new debugging sessions.

TABLE VIII: Study 2 - Breakpoints in the same line of code (Raptor)

Class	**Line of code**	**Breakpoints**
icsUtils	333	3
Game	1751	2
ExamineController	41	2
ExamineController	84	3
ExamineController	87	2
ExamineController	92	2

TABLE IX: Study 1 - Breakpoints in the same line of code (JabRef) in all tasks

Class	Line of code	Breakpoints
SaveDatabaseAction	177,181	4,2
EntryTableTransferHandler	346	2
BasePanel	935	2
BasePanel	969	5
JabRefDesktop	40,84,430	2,2,3
EntryEditor	717,720,721	3,4,2
EntryEditor	723,837,842	2,3,2
EntryEditor	1184,1393	3,2
FieldTextMenu	84	2
JabRefFrame	1119	2
BibtexParser	138,151,159	2,2,2
BibtexParser	160,165,168	3,2,3
BibtexParser	176,198,199,299	2,2,2,2
OpenDatabaseAction	433,450,451	4,2,4
JabRefMain	8	5
AuthorsFormatter	43	5
AuthorsFormatter	131	4
URLUtil	95	2
BibDatabase	175,187,223,456	2,3,2,6

We also analysed if a same class received breakpoints for different tasks. We group all breakpoints by *Class* and count how many breakpoints are set on the type for different tasks, putting “Yes” if a type had a breakpoint, producing Table X. We also count the numbers of breakpoints by type and how many developers set breakpoints on a type.

To Study 1, we observe that ten classes received breakpoints in different tasks by different developers, receiving 77% (160/207) of breakpoints. For example, the type *BibtexParser* had 21% (44/207) of breakpoints in 3 of 5 tasks by *13 different developers*. Note that this analysis only applies to Study 1, since Study 2 only has one task by system, thus does not allow to compare across tasks.

Finally, we count how many breakpoints are in the same method across tasks and participants, indicating that there were “preferred” methods for setting breakpoints, independently of task or participant. We find that 37 methods received at least two breakpoints and *13 methods received five or more breakpoints during different tasks by different developers*, as reported in Figure 3. In particular, the method *EntityEditor.storeSource* received *24 breakpoints* and the method *BibtexParser.parseFileContent* received *20 breakpoints* by different developers on different tasks.

TABLE X: Study 1 - Breakpoints by class across different tasks

Type	Issue 318	Issue 667	Issue 669	Issue 993	Issue 1026	Breakpoints	Dev Diversity
SaveDatabaseAction			Yes	Yes	Yes	7	2
BasePanel	Yes	Yes	Yes		Yes	14	7
JabRefDesktop	Yes	Yes				9	4
EntryEditor			Yes	Yes	Yes	36	4
BibtexParser			Yes	Yes	Yes	44	6
OpenDatabaseAction			Yes	Yes	Yes	19	13
JabRef	Yes	Yes	Yes			3	3
JabRefMain	Yes	Yes	Yes	Yes		5	4
URLUtil	Yes	Yes				4	2
BibDatabase			Yes	Yes	Yes	19	4

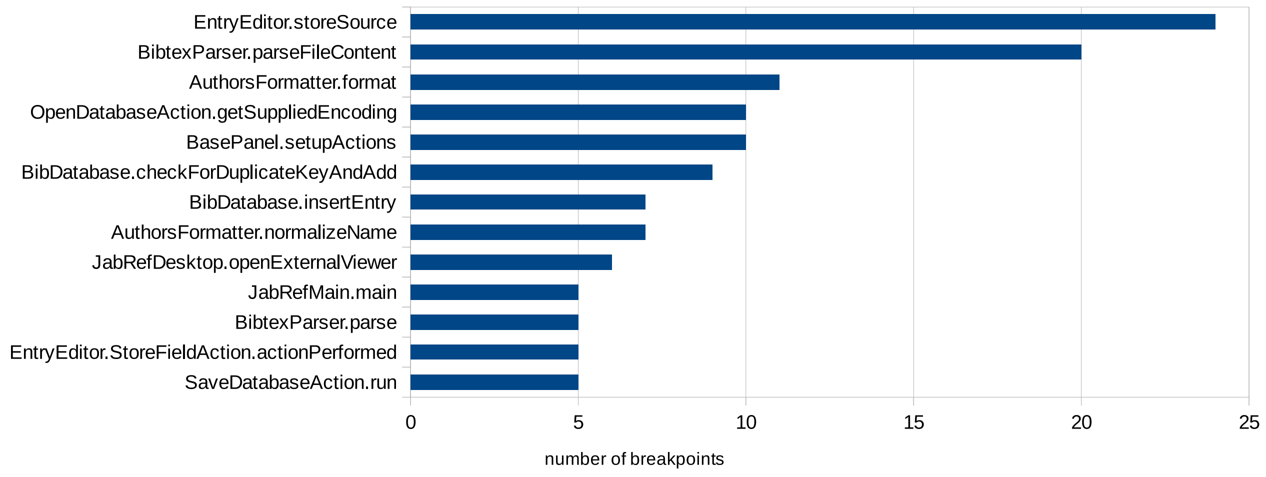


Fig. 3: Methods with 5 or more breakpoints

Our results show that developers do not choose breakpoints lightly, but there is a ***rationale*** *in their setting breakpoints, because different developers set breakpoints on the same line of code for the same task and different developers set breakpoints on the same type or method for different tasks.*

We conclude that there are locations (line of code, class, or methods) on which there were set many breakpoints in different tasks by different developers, showing an opportunity to recommend those locations as candidates for new debugging sessions. However, we could face the bootstrapping problem: we cannot know that these locations are important until developers start to put breakpoints on them. This problem could be addressed with time, by using an infrastructure to collect and share breakpoints. This can enable the accumulation of data that can be used for future debugging sessions. This can encourage more developers to collect and share breakpoints, possibly leading to better automated recommendations.

A surprising result is that different developers, for different tasks, set breakpoints at the same locations. This results shows the usefulness of collecting and sharing breakpoints to assist developers during maintenance tasks.

V. Threats to Validity

Construct Validity threats are related to the metrics used to answer our research questions. We mainly used breakpoints counts and time. Any issue regarding the collection of breakpoints would affect our answers. To mitigate these threats, we carefully studied video captures of the screen of participants and any data extracted from the videos were double-checked by all the authors of the study.

Conclusion Validity threats concern the violations of the assumptions of the statistical tests, and how diverse is the collected data. We reported results regarding percentages of breakpoints set for different kinds of statements, and the common breakpoints set on class/method for same and different tasks. We do not claim any causation relationship between the number/percentage of breakpoints, the kind of statements, and the tasks or developers.

Internal Validity threats are related to the tools used to collect the data and the subject systems. We use SDI and any issue with SDI would affect our results. However, as we validated the collection of breakpoints using the videos, the threat related to SDI is mitigated. We also used videos to identify when developers started and finished the tasks.

In the Study 1, each developer (*e.g.,* freelancer) performed more than one task on the same system. It is possible that a participant may have become familiar with the system after performing earlier tasks and would be knowledgeable enough to set breakpoints when performing the rest of the tasks. However, we didn't observe any significant difference in performance when comparing the performance of same developers for the first and last task.

External validity threats concern the possibility to generalise our results. We share our data and scripts at http://... Further studies with different sets of tasks and different participants are required to confirm our results and generalise our answers.

VI. Related work

We now summarise works related to debugging.

a) Program Understanding: Previous work studied program comprehension and provided tools to support program comprehension. Maalej *et al.* [36] observed and surveyed developers during program comprehension. They concluded that developers need runtime information and reported that developers frequently execute programs using a debugger. Ko *et al.* [37] observed that developers spend large amounts of times navigating between program elements.

Feature and fault location approaches are used to identify and recommend program elements that are relevant to a task at hand [38]. These approaches use defect report [39], domain knowledge [40], version history and defect report similarity [38] while others, like Mylyn [41], use developers' interaction traces, which have been used to study work interruption [42], editing patterns [43], [44], program exploration patterns [45], or copy/paste behaviour [46].

b) Debugging Tools for Program Understanding: Romero *et al.* [47] extended the work by Katz and Anderson [48] and identified high-level debugging strategies, *e.g.,* stepping and breaking execution paths and inspecting variable values. They reported that developers use the information available in the debuggers differently depending on their background and level of expertise.

DebugAdvisor [49] is a recommender system to improve debugging productivity by automating the search for similar issues from the past.

Zayour [8] studied the difficulties faced by developers when debugging in IDEs and reported that the features of the IDE affect the times spent by developers on debugging activities.

c) Automated debugging tools: Automated debugging tools require both successful and failed runs and do not support programs with interactive inputs [50]. Consequently, they have not been widely adopted in practice. Moreover, automated debugging approaches are often unable to indicate the "true" locations of faults [51]. Other more interactive methods, such as slicing and query languages, help developers but, to date, there has been no evidence that they significantly ease developers' debugging activities. However, recent studies showed that empirical evidence of the usefulness of many automated debugging techniques is limited [52]. Researchers also found that automated debugging tools are rarely used in practice [52]. In practice, at least in some scenarios, the time to collect coverage information, manually label the test cases as failing or passing, and run the calculations may exceed the actual time saved using the automated debugging tools.

d) Advanced Debugging Approaches: Zheng *et al.* [53] presented a systematic approach to *statistical debugging* of programs in the presence of multiple bugs, using probability inference and common voting framework to accommodate more general bugs and predicate settings. Ko and Myers [50], [54] introduced *interrogative debugging*, a process with which developers ask questions about their programs outputs to determine what parts of the programs to understand. Pothier and Tanter [55] proposed *Omniscient debuggers*, and approach to support back in time navigation across previous program states. *Delta debugging* [56] by Hofer *et al.* purports that the smaller the failure-inducing input, the less program code is covered. It can be used to minimise a failure-inducing input systematically. Ressia [57] proposed *object-centric debugging*, focusing on objects as the key abstraction execution for many tasks. Estler *et al.* [58] discussed *collaborative debugging* suggesting that debugging collaboration is perceived as important by developers and can improve their experience. This result founded our approach although we use asynchronous debugging sessions.

e) Empirical Studies on Debugging: Jiang *et al.*[3] studied the change impact analysis process that should be done during software maintenance by developers to make sure changes do not introduce new faults. They conducted two studies about change impact analysis during debugging sessions. They found that the programmers in our studies did static change impact analysis before they made changes by using IDE navigational functionalities, and they did dynamic change impact analysis after they made changes by running the programs. We also found that they did not use any change impact analysis tools.

Petrillo *et al.*[35] observed a strong correlation between the time of the first breakpoint ($\rho = -0.637$) and the total elapsed time of debugging activities. The study showed that developers follow different debugging patterns. They also observed that developers who set breakpoints carefully complete tasks faster than developers who toggle breakpoints too quickly.

Zhang *et al.*[5] proposed a method to generate breakpoints based on existing fault localization techniques, showing that the generated breakpoints can usually save some human effort for debugging.

Last but not least, Beller *et al.*. [4] conduct study on how software engineers debug software problems. They performed on-line survey, interviews and collected debugging data using a purpose-built plugin WATCH DOG 2.0. Their results indicate that 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 work is complementary of Beller *et al.*.'s work, exploring different and specific aspects of developers' behaviors on breakpoint usage on which was not discussed previously.

VII. Conclusion

We conducted two observational studies to understand how developers set breakpoints, analysing more than 10 hours of developer's videos in 45 debugging sessions performed by 28 different, independent developers, containing 307 breakpoints from two independent studies on 3 software systems.

Our study results show that (1) setting the first breakpoint is not an easy task and that developers need tools to assist them in locating the places to toggle breakpoints; (2) the time of setting the first breakpoint can be used as a predictor for the duration of debugging tasks; (3) developers did not choose breakpoints simplistically but there is an *underlying rationale* in that decision, because different developers set breakpoints on the same line of code for the same task and different developers toggle breakpoints on the same type or method for different tasks; (4) surprisingly, different, independent developers set breakpoint at same locations for similar debugging tasks and, thus, collecting and sharing breakpoints could assist developers during debugging tasks; (5) 50% (153/307) of breakpoints are toggled on *call* statements and only 2% (7/307) on *while-loop* statements.

We conclude that developers need tools that can assist them in locating adequate places to set breakpoints in the code and our observations suggest the opportunity for a breakpoint recommendation system, similar to previous work [5]. Our observations could also form the basis for building a grounded theory of the developers' use of breakpoints to improve debuggers and other tool support.

In future work, we plan to perform a large scale experiment on interactive debugging, collecting data from several systems and with more participants. We will study the differences between experts and novices as well as professionals and students. We will also study the time for the breakpoints following the first breakpoints as well as the characteristics of the statements where different, independent developers set same breakpoints. Our ultimate goal is to develop a grounded theory of the use of breakpoints to improve the way developers debug. We also want to use developers' breakpoints to recommend breakpoints to other developers.

References

[1] A. S. Tanenbaum and W. H. Benson, "The people's time sharing system," *Software: Practice and Experience*, no. 2, pp. 109–119, apr 1973.

[2] T. D. LaToza and B. a. Myers, "Developers ask reachability questions," *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 1, pp. 185–194, 2010.

[3] S. Jiang, C. McMillan, and R. Santelices, "Do programmers do change impact analysis in debugging?" *Empirical Software Engineering*, pp. 1–39, 2016.

[4] A. Z. Moritz Beller, Niels Spruit, "How developers debug," 2017. [Online]. Available: https://doi.org/10.7287/peerj.preprints.2743v1

[5] C. Zhang, J. Yang, D. Yan, S. Yang, and Y. Chen, "Automated Breakpoint Generation for Debugging," *Journal of Software*, no. 3, pp. 603–616, mar 2013.

[6] R. Tiarks and T. Röhm, "Challenges in Program Comprehension," *Softwaretechnik-Trends*, vol. 32, no. 2, pp. 19–20, May 2013. [Online]. Available: http://link.springer.com/10.1007/BF03323460

[7] K. Araki, Z. Furukawa, and J. Cheng, "A general framework for debugging," *Software, IEEE*, vol. 8, no. 3, pp. 14–20, May 1991.

[8] I. Zayour and A. Hamdar, "A qualitative study on debugging under an enterprise IDE," *Information and Software Technology*, vol. 70, pp. 130–139, feb 2016.

[9] *Debugging Java Applications*, Netbeans. [Online]. Available: https://netbeans.org/project_downloads/usersguide/nbfieldguide/Chapter5-Debugging.pdf

[10] *Managing conditional breakpoints*, Eclipse. [Online]. Available: http://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Ftasks%2Ftask-manage_conditional_breakpoint.htm&cp=1_3_6_0_5

[11] K. Basques, "How to set breakpoints," Google Developers, Tech. Rep., 2017. [Online]. Available: https://developers.google.com/web/tools/chrome-devtools/javascript/add-breakpoints

[12] ——, "Get Started with Remote Debugging Android Devices," Google Developers, Tech. Rep., 2017. [Online]. Available: https://developers.google.com/web/tools/chrome-devtools/remote-debugging/

[13] K. B. Dave Gash, Paul Bakaus, "How to step through your code," Google Developers, Tech. Rep., 2017. [Online]. Available: https://developers.google.com/web/tools/chrome-devtools/javascript/step-code

[14] Mozilla Development Network, "Firefox Developer Tools – Debugger," Tech. Rep., 2016. [Online]. Available: https://developer.mozilla.org/en-US/docs/Tools/Debugger

[15] ——, "Debugging Firefox for Android over USB," Tech. Rep., 2016. [Online]. Available: {https://developer.mozilla.org/en-US/docs/Tools/Remote_Debugging/Debugging_Firefox_for_Android_with_WebIDE}

[16] The Ecipse Foundation, "Eclipse documentation – Running and Debugging," Tech. Rep., 2016. [Online]. Available: http://help.eclipse.org/neon/topic/org.eclipse.jdt.doc.user/tasks/task-add_line_breakpoints.htm

[17] ——, "Eclipse documentation - Remote Debugging," Tech. Rep., 2016. [Online]. Available: http://help.eclipse.org/neon/topic/org.eclipse.jdt.doc.user/concepts/cremdbug.htm

[18] JetBrains, "Enabling, Disabling and Removing Breakpoints," Tech. Rep., 2016. [Online]. Available: https://www.jetbrains.com/help/idea/2016.3/enabling-disabling-and-removing-breakpoints.html

[19] ——, "Remote Debugging," Tech. Rep., 2016. [Online]. Available: https://www.jetbrains.com/help/idea/2016.3/remote-debugging.html

[20] Netbeans, "NetBeans IDE Field Guide, Debugging Java Applications," Tech. Rep., 2016.

[21] ——, "NetBeans IDE Field Guide, Debugging Java Applications," Tech. Rep., 2016.

[22] ——, "NetBeans IDE Field Guide, Debugging Java Applications," Tech. Rep., 2016.

[23] Microsoft, "Using Breakpoints," Tech. Rep., 2016. [Online]. Available: https://msdn.microsoft.com/en-us/library/5557y8b4.aspx

[24] ——, "Remote Debugging," Tech. Rep., 2016. [Online]. Available: https://msdn.microsoft.com/en-us/library/y7f5zaaa.aspx

[25] Python, "The Python Debugger," Tech. Rep., 2017. [Online]. Available: https://docs.python.org/3/library/pdb.html

[26] ——, "remote-pdb 1.2.0," Tech. Rep., 2017. [Online]. Available: https://pypi.python.org/pypi/remote-pdb

[27] Apple Developer, "Debugging Tools," Tech. Rep., 2017. [Online]. Available: https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/debugging_with_xcode/chapters/debugging_tools.html

[28] Richard Stallman, Roland Pesch, Stan Shebs, "The GNU Source-Level Debugger – Debugging Tools," GNU, Tech. Rep., 2002. [Online]. Available: ftp://ftp.gnu.org/old\-gnu/Manuals/gdb-5.1.1/html_node/gdb_toc.html

[29] Oracle, "jdb the java debugger," Tech. Rep., 2016. [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html

[30] ——, "Java debugger (jdb)," Tech. Rep., 2016. [Online]. Available: http://www.ibm.com/support/knowledgecenter/SSYKE2_7.0.0/com.ibm.java.win.70.doc/user/jdb.html

[31] Android Studio, "Debug Your App," Tech. Rep., 2016. [Online]. Available: https://developer.android.com/studio/debug/index.html

[32] ——, "Android Debug Bridge," Tech. Rep., 2016. [Online]. Available: https://developer.android.com/studio/command-line/adb.html

[33] Cloud 9, "Debugging Your Code," Tech. Rep., 2016. [Online]. Available: https://docs.c9.io/docs/debugging-your-code

[34] ——, "Debugger Proxy," Tech. Rep., 2016. [Online]. Available: https://cloud9-sdk.readme.io/docs/debugger-proxy

[35] F. Petrillo, Z. Soh, F. Khomh, M. Pimenta, C. Freitas, and Y.-G. Guhneuc, "Towards understanding interactive debugging," in *In Proceedings of the 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, August 2016, p. 10.

[36] W. Maalej, R. Tiarks, T. Roehm, and R. Koschke, "On the comprehension of program comprehension," *ACM Transactions on Software Engineering and Methodology*, vol. 23, no. 4, pp. 31:1–31:37, Sep. 2014. [Online]. Available: http://doi.acm.org/10.1145/2622669

[37] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transaction on Software Engineering*, vol. 32, no. 12, pp. 971–987, dec 2006.

[38] S. Wang and D. Lo, "Version history, similar report, and structure: putting them together for improved bug localization," in *Proceedings of the 22nd International Conference on Program Comprehension - ICPC 2014*. New York, New York, USA: ACM Press, 2014, pp. 53–63.

[39] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, jun 2012, pp. 14–24.

[40] X. Ye, R. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*. New York, New York, USA: ACM Press, 2014, pp. 689–699.

[41] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, 2006, pp. 1–11.

[42] H. Sanchez, R. Robbes, and V. M. Gonzalez, "An empirical study of work fragmentation in software evolution tasks," in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, 2015, pp. 251–260.

[43] A. Ying and M. Robillard, "The influence of the task on programmer behaviour," in *Proceedings International Conference on Program Comprehension*, june 2011, pp. 31–40.

[44] F. Zhang, F. Khomh, Y. Zou, and A. E. Hassan, "An empirical study of the effect of file editing patterns on software quality," in *Proceedings Working Conference on Reverse Engineering*, 2012, pp. 456–465.

[45] Z. Soh, F. Khomh, Y.-G. Guéhéneuc, G. Antoniol, and B. Adams, "On the effect of program exploration on maintenance tasks," in *Reverse Engineering (WCRE), 2013 20th Working Conference on*, Oct 2013, pp. 391–400.

[46] T. M. Ahmed, W. Shang, and A. E. Hassan, "An empirical study of the copy and paste behavior during development," in *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, May 2015, pp. 99–110.

[47] P. Romero, B. du Boulay, R. Cox, R. Lutz, and S. Bryant, "Debugging strategies and tactics in a multi-representation software environment," *International Journal of Human-Computer Studies*, vol. 65, no. 12, pp. 992–1009, dec 2007.

[48] I. Katz and J. Anderson, "Debugging: An Analysis of Bug-Location Strategies," *Human-Computer Interaction*, vol. 3, no. 4, pp. 351–399, dec 1987.

[49] B. Ashok, J. Joy, H. Liang, S. K. Rajamani, G. Srinivasa, and V. Vangala, "DebugAdvisor: A Recommender System for Debugging," in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering on European software engineering conference and foundations of software engineering symposium - E*. New York, New York, USA: ACM Press, 2009, p. 373.

[50] A. Ko, "Debugging by asking questions about program output," *Proceeding of the 28th international conference on Software engineering - ICSE '06*, p. 989, 2006.

[51] J. Rößler, "How helpful are automated debugging tools?" in *2012 1st International Workshop on User Evaluation for Software Engineering Researchers, USER 2012 - Proceedings*, no. Section V, 2012, pp. 13–16.

[52] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" *Proceedings of the 2011 International Symposium on Software Testing and Analysis ISSTA 11*, p. 199, 2011.

[53] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken, "Statistical Debugging : Simultaneous Identification of Multiple Bugs," *Challenges*, vol. 148, pp. 1105–1112, 2006.

[54] A. Ko and B. A. Myers, "Finding Causes of Program Output with the Java Whyline," in *CHI 2009 - Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM, Ed., New York, New York, USA, 2009, pp. 1569–1578.

[55] G. Pothier and É. Tanter, "Back to the Future: Omniscient Debugging," *IEEE Software*, vol. 26, no. 6, pp. 78–85, nov 2009. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5287015

[56] B. Hofer and F. Wotawa, "Combining slicing and constraint solving for better debugging: the CONBAS approach," *Advances in Software Engineering*, vol. 2012, p. 13, 2012.

[57] J. Ressia, A. Bergel, and O. Nierstrasz, "Object-centric debugging," in *Proceedings - International Conference on Software Engineering*, 2012, pp. 485–495.

[58] H. C. Estler, M. Nordio, C. a. Furia, and B. Meyer, "Collaborative debugging," *Proceedings - IEEE 8th International Conference on Global Software Engineering, ICGSE 2013*, pp. 110–119, 2013.