How Do Developers Toggle Breakpoints?
–An Empirical Study–

Fabio Petrillo, Hyan Mandian, Aiko Yamashita, Foutse Khomh, Yann-Gaël Guéhéneuc
Polytechnique Montréal, QC, 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 use debugging to fix faults and also implementing new features. They spend at least 30% of their time on debugging. They use interactive development environments, like Eclipse, to perform their debugging activities. Yet, to the best of our knowledge, very few studies investigated how developers spend time and efforts during interactive debugging sessions. In particular, when starting a new interactive debugging session, developers must spend time and effort in finding where to toggle useful breakpoints. Using the Swarm Debugging Infrastructure (SDI), we report our analyses of debugging-related data collected with 20 developers (12 students and 8 professional freelancers) debugging five bugs found in the open-source project JabRef. We collected more than 6 hours of developer's debugging data containing 207 breakpoints. Results of our analyses show that, first, developers choose breakpoints purposefully; second, breakpoint toggling is a hard task, especially for the first breakpoint. We conclude that developers need tools that can assist them in locating places to toggle breakpoints in the code. These results show the potential benefits of sharing debugging activities to support software maintenance and evolution. Finally, we discuss some implications of our results for tool developers and future debuggers.**

I. Introduction

Debugging is a recurring and important activity during software development, maintenance, and evolution [1]. During debugging, developers use debuggers to detect, locate, and correct faults and–or implement new features. Interactive debugging is one particular debugging process in which developers use tools to investigate the execution of a program interactively. Modern debuggers are often integrated in development environments, *e.g.,* DDD [2] or the debuggers of Eclipse, Netbeans, Intellij IDEA, and Visual Studio. Debuggers allow navigating through the code and toggling breakpoints to locate and correct faults and–or implement new features.

Latoza *et al.* [3] showed that developers spend at least 30% of their time on debugging. Debugging is a time-consuming activity and improving the developers' performance during debugging could increase their productivity.

Tiarks and Röhms [4] observed that developers set a lot of breakpoints at the beginning of their debugging activities to discard irrelevant breakpoints as they debug. Setting breakpoint is a difficult activity and supporting developers in setting "right" breakpoints could also improve their productivity.

In previous work [5], we observed a strong correlation between the time of the first breakpoint ($\rho = -0.637$) and

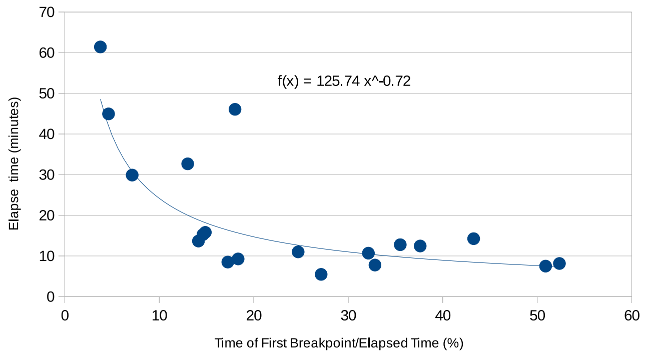

Fig. 1: Relation between time of first breakpoint and task elapsed time

the total elapsed time of debugging activities, as reported on Figure 1. Setting the first breakpoint is representative of developers' understanding of the bug and conditions their productivity.

Maalej *et al.*[6] observed that capturing contextual information requires the instrumentation of the IDE and continuous observation of the developers' activities within the IDE. Recent studies by Storey *et al.* [7] showed that the newer generation of developers, who is proficient in social media, is comfortable with sharing such information. Developers are nowadays opened, transparent, eager to share their knowledge, and generally willing to allow information about their activities to be collected by the IDE automatically. So, why shouldn't developers share their debugging activities?

We answer this question by introducing the concept of Swarm Debugging to support a developer's debugging activities using information collected from other developers' debugging activities. Swarm Debugging uses contextual information obtained through the instrumentation of the debugger by the Swarm Debugging Infrastructure (SDI). The SDI also provides interactive tools and visualizations to share this data among developers.

The concept of Swarm Debugging is based on the idea that many developers, performing independently debugging activities, are in fact building collective knowledge. Thus, developers need support to collect, store, and share information from and about their debugging activities, including but not limited to breakpoint locations, visited statements, and traversed paths, which establish a context for current and future debugging activities by these and other developers.

Yet, to the best of our knowledge, there exist no study of the habits in breakpoint settings of a set of participants fixing the same set of bugs although such study would inform researchers and tool builders to support developers during their debugging activities.

Consequently, we report a case study to understand where developers set breakpoints when debugging five real bugs found in the JabRef Java open-source program. Our study involves 20 developers (12 students and eight professional freelancers).

We collect more than 6 hours of data related to their debugging session activities (breakpoint setting and statement/invocations stepping-ins and -overs, including 207 breakpoints).

Using our data and its analyses, we answer the following three research questions about the characteristics of breakpoints:

RQ1: How much time do developers spend in a debugging session before toggling their first breakpoint?

With this research question, we want to quantify in time this effort. We find that, in average, participants toggle their first breakpoint after 27% of their debugging activity time.

RQ2: On what kind of statement do developers toggle their breakpoints?

This question has for objective to observe if participants choose predominantly a kind of statement. We find that 53% (111/207) of all toggled breakpoints are on *call* statements and only 1% (3/207) on *while-loop* statements.

RQ3: Do different developers toggle breakpoints at the line of code, type or method for the same task?

We investigate if the locations of the breakpoints are random or not by analysing exactly (line of code) where each breakpoint was toggled. We report that 39 out of 207 breakpoints (about 20%) are toggled at *exactly* the same lines of code when different participants fix the *same* bugs. Alse, we want to observe if participants toggle breakpoints simplistically or they chose rationally at the type level. We observe that 10 types received 77% (160/207) of all the breakpoints for different bugs by different developers. Finally, we want to observe if participants have "preferred" methods in which toggling breakpoints, independently of bugs or participant. We report that 37 methods received at least two breakpoints and *13 methods* received five or more breakpoints.

Our results show that developers do not choose breakpoints simplistically and that there is a **rationale** in their setting breakpoints. We also show that for a same bug, different developers set the same breakpoints. Thus, breakpoints could be recommended across bugs and developers.

The remainder of the paper is organized as follows. Section II provides some background about debugging and describes Swarm Debugging. Section III presents the design of our experiment. Sections IV-A, IV-B, IV-C show and discuss the results of our research questions. Section V discuss the 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 debugging, Swarm Debugging and the Swarm Debug Infrastructure (SDI).

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.*describe the debugging activity as an interactive process where developers make hypotheses and verify them by examining problems in a program [8]. Developers then refute or validate their hypotheses until the problems are resolved.

Interactive debugging consists in using a debugger tool to detect, locate, and correct a fault in a program. A process also known as *program animation*, *stepping*, or *following execution* [9]. Developers often refer to this process as simply *debugging*, because several IDEs provide debuggers to support *debugging*. However, while *debugging* is the process of finding faults, *interactive debugging* is one particular debugging approach in which developers use tools to investigate the execution of a program interactively. We use the expressions *interactive debugging* or *stepping*, but there is not yet a consensus on what is the best name for this debugging process.

B. Foundations of Swarm Debugging

Software systems are nowadays usually large and complex. Research works in software engineering provide approaches to manage this complexity, regardless of the software development processes, be them "traditional", i.e., systematic waterfall-like processes, or "modern", i.e., agile processes [10]. Recent research works recognize the fundamental importance of crowd-sourcing in software engineering [11]. Swarm Debugging is crowd-sourcing applied to debugging activities and inspired by the combination of *swarm intelligence* and *information foraging theory*.

a) Swarm intelligence (SI): describes the behavior resulting from the self-organization of social insects [12]. Self-organization is the set of dynamic mechanisms enabling structures to appear at the global level of a system from interactions among its lower-level components, without being explicitly coded at the lower-levels.

Ant nests and the societies that they house are examples of SI [13]. Individual ants can only perform relatively simple activities, the whole colony can collectively accomplish sophisticated movement patterns. Ants achieve SI using information encoded as chemical signals (pheromone) deposited by other ants, e.g., indicating a path to follow or an obstacle to avoid.

Similarly, large and complex software systems could be also examples of SI. Individual developers usually can perform activities without having a global understanding of the whole systems [14]. Developers achieve SI using information encoded in software artefacts, most importantly source code but also by-products produced during the development of software, and shared through version-control systems. More, many developers working independently by executing different tasks and interacting locally could produce human-based complex adaptive patterns.

In a bird's eye view, software development is analogous to some animal collective behavior and SI in which a group agents interacting locally with one another and with their environment, following simple rules, and whose interactions lead to the emergence of a global behavior, previously unknown/impossible by individual agents. We claim that the similarities between the self-organization of ant nests and a complex software systems is not a coincidence. Cockburn suggested that the best architectures, requirements, and designs emerge from self-organizing developers, growing in steps and following their changing knowledge, and the changing wishes of the user community [15].

b) Information foraging theory (IFT): is based on the optimal foraging theory developed by Pirolli and Card [16] to understand how people search for information. Optimal foraging theory is rooted in biology and studies and theories of how animals hunt for food. Lawrance et al.[16] extended the concept and applied IFT to support debugging.

However, no previous work proposed the sharing of knowledge related to fine-grained debugging activities. Differently from work [16] that uses IFT on a model *one prey/one predator*, our approach models *many developers* working independently in many *debugging activities* and sharing information to allow SI to emerge. Debugging becomes a foraging process in a swarm intelligence environment.

These concepts - SI and IFT - have led to the design of a crowd-sourcing approach applied to debugging activities: a different, collective way to doing debugging that collects, shares, retrieves information from debugging sessions in order to support (current and future) debugging activities.

To evaluate SD, we have built an infrastructure to support it: **Swarm Debug Infrastructure** (see https://github.com/SwarmDebugging). The Swarm Debug Infrastructure (SDI) [17] implements the SD approach, providing a set of tools for collecting, storing, sharing, retrieving, and visualizing data collected during developers' debugging activities.

Swarm Debugging works as follows. First, several developers perform their individual, lonely debugging activities. During debugging, debugging events are collected by listeners (Label A in Figure 2), for example breakpoints-toggling and stepping events (Label B in Figure 2), that are then stored in a debugging-knowledge repository (Label C in Figure 2). To access this repository, services are defined and implemented in SDI. For example, stored events are processed by dedicated algorithms (Label D in Figure 2) (1) to create (several types of) visualizations, (2) to offer (distinct ways of) searching, and (3) to provide recommendations to assist developers during debugging. Recommendations can pertain to the locations where to toggle breakpoints. Storing and using these events allow to share developers' knowledge among developers, creating a collective intelligence about the software systems and their debugging.

We chose to instrument the Eclipse IDE, a popular IDE, to implement Swarm Debugging and to reach a great number of users. Also, we use services in the cloud to collect the debugging events and to process these events and to provide visualizations and recommendations from these events.

During debugging, developers analyze the code, toggling breakpoints and stepping in and through statements. While traditional dynamic analysis approaches collect all interactions, states or events, SD collects only invocations explicitly explored by developers: SDI collects only visited areas and paths (chains of invocations by *e.g.,Step Into* or F5 in Eclipse IDE) and, thus, does not suffer from performance or memory issues as omniscient debuggers [18] or tracing-based approaches could.

We have defined domain concepts to model software projects and debugging data in Swarm Debugging approach. This meta-model has two main goals. First, it represents the conceptual model of the SD approach. By definition, a conceptual model is a mapping of the concepts and relations of a domain, reflecting the real-world relationships and dependencies. Thus, the meta-model summarises the central concepts adopted in SD. Second, it presents the essential elements necessary to build an infrastructure for SD.

The concepts are inspired and complement the simplified FAMIX Data model[19] with debugging data. FAMIX exploits meta-modelling techniques to make the data model extensible. The simplified view of the FAMIX data model comprises the main object-oriented concepts namely Type, Method, plus the necessary associations between them namely Invocation and Access.

The Swarm Debugging meta-model concepts (Fig. 3) include:

- **Developer** is an SD user, which creates and executes debugging sessions.
- **Product** is the target software product. Product is a set of source code projects (one or more).
- **Task** is a task to be executed by developers, like software comprehension, bug location, software maintenance or refactoring.
- **Session** represents a Swarm Debugging session. It relates developer, project, and debugging events.
- **Type** represents classes and interfaces in the project. Each type has a source code and a file. To simplify,

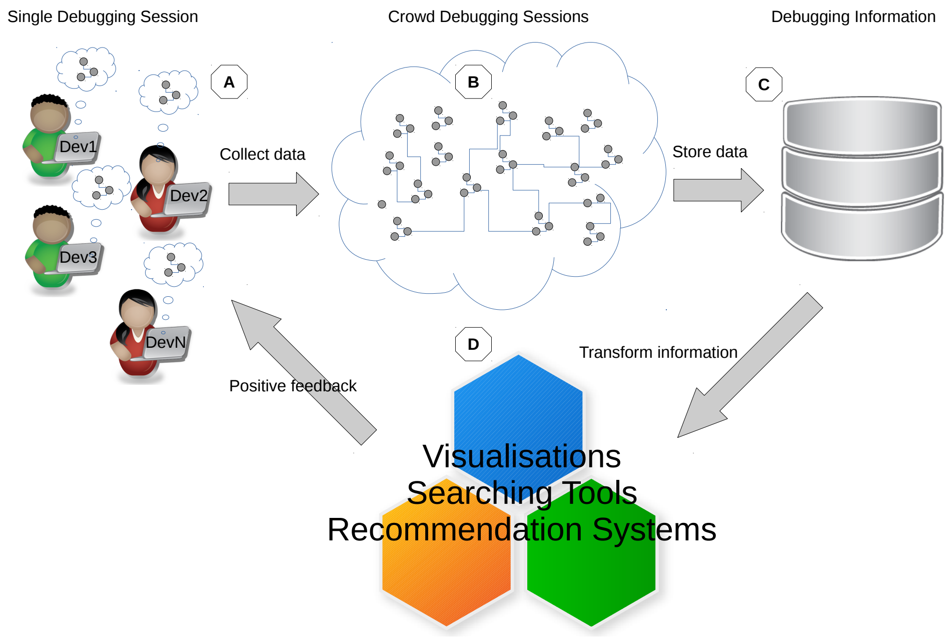

Fig. 2: Overview of the Swarm Debugging approach

SD only considers types that have source code available as belonging to the project domain, ignoring external libraries.

- **Method** is a method, procedure or function associated with a type, which can be invoked during debugging sessions.
- **Namespace** is a container for types. In Java, for example, namespaces are declared with the keyword *package*.
- **Invocation** is a method invoked from another method. It is formed by a pair of methods: an invoking (caller) and an invoked (called).
- **Breakpoint** represents the data collected when a developer toggles a breakpoint in an IDE. Each breakpoint is associated with a type and a method if appropriate.
- **Event** is an event data that is collected when a developer performs some actions during a debugging session, typically stepping events (step into a method, step over, run, return to the caller, *e.g.,*).

C. *The Swarm Debug Infrastructure*

The Swarm Debug Infrastructure (SDI) is an implementation of Swarm Debugging, providing tools for collecting, sharing, and retrieving debugging data collected during developers' debugging sessions. SDI uses a data frugality approach [20], collecting only paths that were **intentionally explored** by developers, collecting only methods **explicitly visited** by developers. It means that we collect only invocations where developers call a method and intentional events (*e.g., Step Into* or F5 in Eclipse IDE) for visiting a method invoked by an analysed method.

SDI can complement other infrastructures like *Mylyn*[21], *Hipikat*[22][23] and *DebugAdvisor* [24], with debugging session data (either structured or unstructured), and analysing fine-grained events to collect breakpoints or information about debugged software areas. SDI, Hipikat, and DebugAdvisor share the same essential idea: using previous data to support debugging tasks. Moreover, SDI and Mylyn share the idea of considering context-awareness in their activities.

III. Experiment Design

This section presents the design of our experiment to address the following three research questions:

RQ1: How much time do developers spend in a debugging session before toggling their first breakpoint?

RQ2: On what kind of statement do developers toggle their breakpoints?

RQ3: Do different developers toggle breakpoints at the line of code, type or method for the same task?

To answer the research questions, we proceeded as follows [1]:

A. *Tasks*

We use debugging tasks to start participants' debugging sessions. We choose to ask participants to find the locations of true faults in an independent, open-source program. We select JabRef[2] as object program. JabRef is a bibliography reference manager developed in Java. It has faults publicly reported in its issue tracker, its domain is easy to understand, and it is composed of relatively independent packages and classes (high cohesion, low coupling). We selected carefully five faults reported against JabRef v3.2 in its issue tracker, choosing tasks on which developers should have to stepping

[1]Artifacts are available on onhttp://swarmdebugging.github.io/publications

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

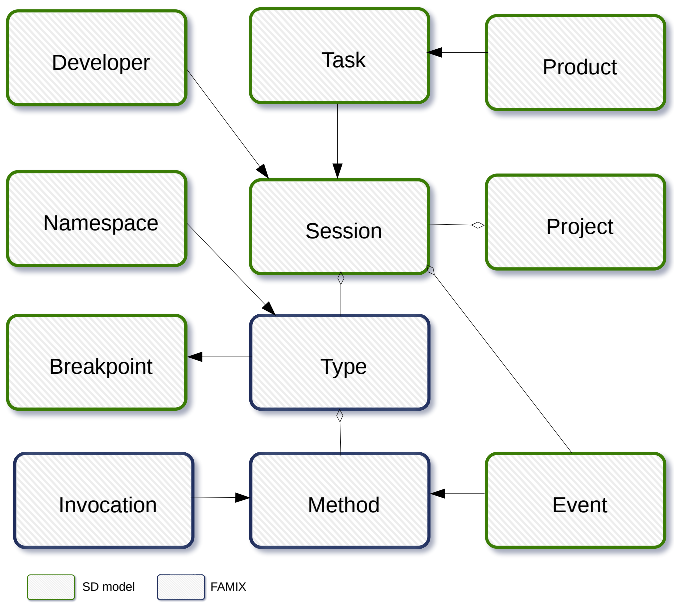

Fig. 3: The Swarm Debugging meta-model

Task	Average Times (min.)
318	13
667	31
669	11
993	28
1026	21

TABLE I: Elapse time by task (average)

different Java classes (from the JabRef user interface layer) to achieve the fault. Thus, we ask participants to find the locations of the faults described in issues 318, 667, 669, 993, and 1026 on JabRef issue's track.

We estimate the developers' effort on each task a posteriori by calculating the averages of the elapse times for each task by participants. Table I shows the average times (in minutes) for each task. Participants spend **21 minutes** to complete the tasks in average. We believe that this amount of time is reasonable and representative of simple bug fixing while inducing little fatigue in participants.

B. Participants

We strive to have a sample of participants that is realistic and representative of the industry. We recruit eight professional developers through a freelancer-hiring Web site[3]. All are male. Two are experts and three are intermediate in Java. They all use Eclipse and its debuggers frequently. We also ask volunteers among our undergraduate and graduate students at Polytechnique Montréal to participate in our experimental study and 12 students volunteered. They are expert or advanced developers (70%). They all used IDEs (70%) and debuggers (60%) frequently. Hence, the participants are representative of junior and established developers in early or mid-careers. We collect all participant profile data from a survey applied before the experiment. All data about expertise are informed by them, but we confirmed all participants had enough expertise to participate in the experiment because we used video session recordings to evaluate their level of expertise.

C. Artifacts

We provide participants with two documents. The first document is a tutorial (http://swarmdebugging.org/publications/experiment/tutorial.html)explaining how to install and configure the tools required for the experiment and how to carry a warm-up task and the experiment. The warm-up task is presented using a video that guides the participants[4]. The second document presents the five issues with a description and some steps to reproduce the faults. Again, we offer a video demonstrating step-by-step how to reproduce the faults to reduce the participants' efforts.

We also provide a preconfigured Eclipse workspace to participants and ask them all to install Java 8, Eclipse Mars 2, the Swarm Debug Tracer plug-in v0.1. The Eclipse workspace contain two Java projects: a Tetris game for the warm-up task and JabRef v3.2 for the experiment. We use the warm-up task on the Tetris game to confirm that the participants' environments are correctly configured and that the participants understand the experiment. Also, we require participants to install and configure the Open Broadcaster Software[5] (OBS), an open-source system for live streaming and recording.

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

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

[5]https://obsproject.com

We finally administer a post-experiment on-line questionnaire to the participants to collect information about the experiment, including the follow questions:

1) Did you find the bug/issue?
2) Where is the bug/issue?
3) Why does the bug/issue happen?
4) Were you tired?
5) Describe your debugging experience.

D. Experiment Procedure

After installing their environments, participants perform the warm-up task, which consists of starting a debugging session, toggling a breakpoint, and debugging the Tetris program to locate a given method. It was a truly trivial task that we used to filter the participants. All participants who performed our experiment executed correct the warm-up task [6]

After performing the warm-up task and an authentication process, each participant realised debugging sessions to find the locations of the five faults. They are informed that were participating in a research experiment about debugging, but we did not inform exactly what data are doing collect nor how. We limited on one-hour maximum by task, but we suggested participants spend about 20 minutes by a fault. We asked participants to control their fatigue and move on to the next task if they feel tired.

All debugging data (breakpoints, stepping, method invocations) were automatically and transparently collected by the Swarm Debug Tracer and stored on our SDI Services. We collect this data in the course of 8 days. We also collect the video capture of the participant's screens during their debugging sessions. We thus obtain:

- Video captures, one per participant per task using OBS. The videos are essential to control each execution quality, and producing a reliable and reproducible evidence on our results.
- The statements on which the participants' toggled breakpoints. We collect the real statement, from the line of text in the code source, analysed during each breakpoint creation (monitoring by the Swarm Debug Tracer). We consider the following categories of statements[7]:
  - *call*
  - *if-statement*
  - *assignment*
  - *return*
  - *while-loop*.

We thus can compute, for each participant on each task:

- Start Time (ST): the effective time when a developer starts a task. We analysed each video, and we started to counting when effectively the developer start a task (when she start the Swarm Debug Session for example). It means that we did not start to counting on time "00" of the video, but only one starts the task.
- Time of First Breakpoint (FB): the time when a developer toggles the first breakpoint.
- End time (T): the effective time when a developer finishes a task.
- Elapse End time (ET): $ET = T - ST$
- Elapse Time First Breakpoint (EF): $EF = FB - ST$

After their debugging sessions, participants fill the on-line questionnaire and provide the video captures. We control whether participants were successful or not comparing their answers in the questionnaire, the video recording and the true bug locations because we know each fault point (all tasks were solved by JabRef's developers).

IV. Experiment Results

We now present the data collected during our experiment and the results of our analyses. We collect 28 video captures, for more than 6 hours of developers' activities. We have 38 debugging sessions by 20 developers, 207 breakpoints toggled during the sessions, and more than 6,000 method invocations. We used those data to answer follow research questions.

A. RQ1: How much time do developers spend in a debugging session before toggling their first breakpoint?

We normalise the times elapsed between the start of a debugging session and the setting of the first breakpoint (MFB) by the total duration of the task to compare tasks and participants:

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

We find that, in average, **participants spend 27% of task time to toggle the first breakpoint** (std. dev. 17%), *i.e.,* about 1/4 of the participants' times are used to locate where to toggle their first breakpoints.

We conclude that toggling the first breakpoint is not an easy task and developers need tools to assist them in locating the places to toggle breakpoints.

B. RQ2: On what kind of statement do developers toggle their breakpoints?

We classify the kinds of statements on which the participants toggled their breakpoints. We analysed each breakpoint and obtained Table II, which shows that 53% (111/207) of breakpoints are toggled on **call statements** and only 1% (3/207) on while-loop statements.

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

TABLE II: Breakpoints by kind of statement

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

[7]https://en.wikipedia.org/wiki/Statement_(computer_science)

After grouping *if-statement*, *return*, and *while-loop* into *control-flow* statements, we report in Figure 4 that 29% of breakpoints are on control-flow statements, about 1/4. Developers prefer *call* statements because they would like to analyse the software state before coming in a method. Furthermore, *call* statements represent the behaviour of objects, while the *control-flow* statements represent the behaviour of methods, one level of abstraction lower. That result can be useful, for instance, when debugger's developers build a new breakpoints' recommendation system who could use that result as a heuristic to prioritise *call* statements to suggest breakpoints.

*We results show that 53% (111/207) of breakpoints are toggled on **call statements** and only 1% (3/207) on while-loop statements.*

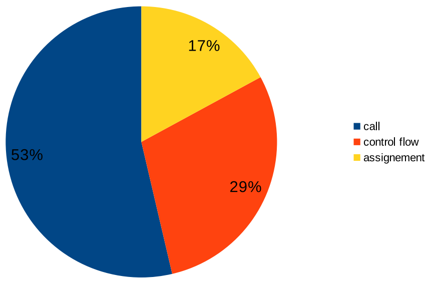

Fig. 4: Breakpoints by kind of statement - call, control flow and assignment

C. RQ3: Do different developers toggle breakpoints at the line of code, type or method for the same task?

We investigate each breakpoint to assess whether there are breakpoints on the same line of code for different participants, analysing breakpoints on the same task and on different tasks. We group all breakpoints by **task** and count how many breakpoints are toggled on the same line of code several times for each task across participants. We reported the results in Table III. We observe that 39 breakpoints out of 207 are toggled in the **exactly** same line of code for the same task toggled by different developers. That result shows that developers do not choose breakpoints simplistically, as suggests [4], but there is a **rational** in that decision because different developers toggle the same line of code for the same task[8].

We also analysed if a type had breakpoints for different tasks. Thus, we group all breakpoints by **type** and count how many breakpoints are toggled on the type for different tasks, putting "1" if a type had a breakpoint, producing Table IV. We also count the numbers of breakpoints by type and how many developers toggle breakpoints on a type.

We observe that ten types received breakpoints in different tasks by different developers, receiving 77% (160/207) of toggle breakpoints. For example, the type **BibtexParser** had 21% (44/207) of toggle breakpoints in 3 of 5 tasks by **13 different developers**.

[8]In fact, that number could be higher if we have to consider a region (a threshold) of code and not only a line of code. We chose counting only the same line of code because it is an evidence that line was important for debugging a task.

Task	Type	Line	# same line
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 III: Breakpoints in the same line of code by task

Finally, we count how many breakpoints are in the same method across tasks and participants, indicating that there were "preferred" methods for toggling breakpoints, independently of task or participant. We find 37 methods received at least two breakpoints and **13 methods that receive five or more breakpoints during different tasks by different developers**, as reported in Figure 5. In special, the method *EntityEditor.storeSource* received **24 breakpoints** and the method *BibtexParser.parseFileContent* received **20 breakpoints** by different developers on different tasks.

*Our results show that developers did not choose breakpoints simplistically, but there is a **rational** in that decision, because different developers toggle the same line of code for the same task and different developers toggle the same type or method for different tasks.*

That conclusion has an important implication: there are places (line of code, type, or methods) that were toggled many breakpoints in different tasks by several developers, showing an opportunity to use those places as candidates for new debugging sessions. In another hand, one could arguments that we have a bootstrapping problem: we cannot know that these methods are important until developers start to put breakpoints in them. However, that issue is addressed by the time, because using the Swarm Debug Infrastructure has a collaborative approach (Swarm Debugging), on which previous debugging data are accumulated, forming a dataset to be used on new debugging sessions. Consequently, more developers use SDI, and more they will improve the dataset about a software system, closing a feedback positive cycle.

Type	0318	0667	0669	0993	1026	Breakpoints	Dev Diversity
SaveDatabaseAction	0	0	1	1	1	7	2
BasePanel	1	1	1	0	1	14	7
JabRefDesktop	1	1	0	0	0	9	4
EntryEditor	0	0	1	1	1	36	4
BibtexParser	0	0	1	1	1	44	6
OpenDatabaseAction	0	0	1	1	1	19	13
JabRef	1	1	1	0	0	3	3
JabRefMain	1	1	1	1	0	5	4
URLUtil	1	1	0	0	0	4	2
BibDatabase	0	0	1	1	1	19	4

TABLE IV: Breakpoints by type in different tasks

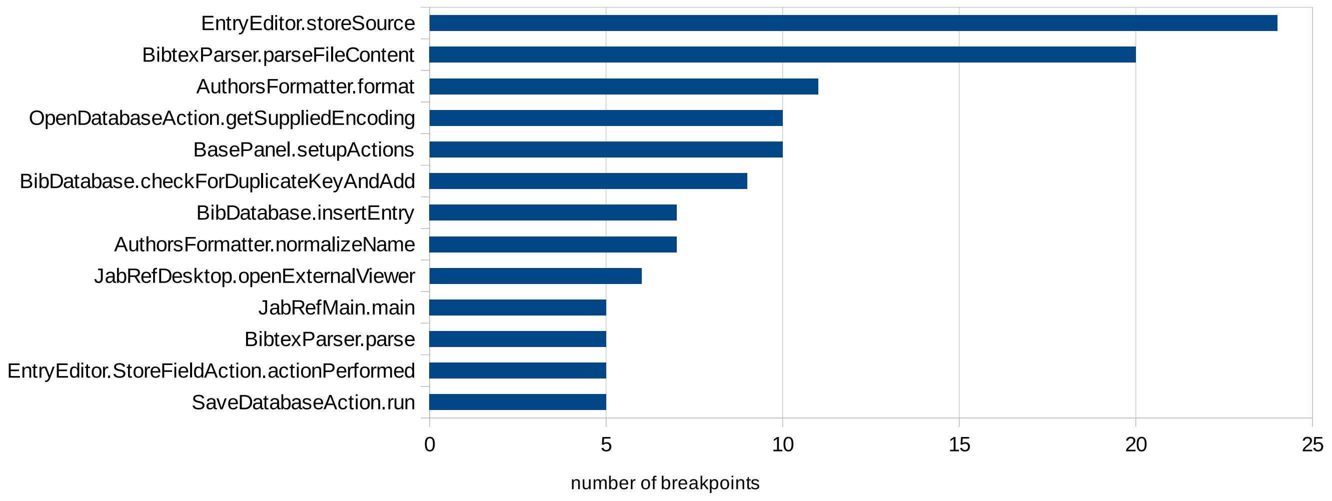

Fig. 5: Methods with 5 or more breakpoints

V. Threads to Validity

As our experiment involved both students and professional freelancer developers, the way the two kinds of participants debug programs may be varied. Professional freelancer developers may tend to carefully and methodically toggle breakpoints while students may not. Thus, we run our prediction approach on the whole dataset, then on only freelancer dataset in one and, and only on students dataset on the other hand. This distinction allows us to access whether the debugging activities of one kind of participant could be more "rich" for breakpoint prediction than other.

Our results are subject to threats to their validity as any other experiment study.

Construct Validity threats is related to the metrics used to answer our research questions. We mainly used breakpoints counts which are a precise measure. However, we considered the breakpoints collected by our swarm debugging infrastructure (SDI). Any issue regarding the collection of breakpoints with SDI would affect our results. To mitigate these threats, we collected both SDI data and video captures of the screen of participants. We compared information extracted from the videos with the data collected by SDI and found that the breakpoints collected by SDI are exactly those toggled by developers.

Conclusion Validity threats concerns the violations of the assumptions of the statistical tests, and how diverse is the collected data. We reported results regarding percentages of breakpoints toggled for different kinds of statements, and the common breakpoints toggled on class/method for the same and different tasks. We did not perform any statistical analysis to answer our research questions. Thus, our results do not suffer from any statistical assumptions. 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 start and finish the tasks. We use only one system in our study (*i.e.,* JabRef). We performed our study on a single system because we needed to have enough data points from a single system to assess the effectiveness of breakpoint prediction. We should collect more data on other systems and check whether the system used can affect our results. 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 toggle breakpoints when performing later 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://swarmdebugging.org/publications/icsme2016. Further studies with different sets of tasks and different participants are required to verify our results and make our findings more general.

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.* [25] 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.* [26] observed that developers spend large amounts of times navigating between program elements. Sillito *et al.* identified the questions that developers ask when finding and extending starting methods [27]. They described how developers answer these questions during software maintenance activities.

Feature and fault location approaches are used to identify and recommend program elements that are relevant to a task at hand [28]. These approaches use bug report [29], domain knowledge [30], version history and bug report similarity [28] while others, like Mylyn [31], use developers' interaction traces, which have been used to study work interruption [32], editing patterns [33], [34], program exploration patterns [35], or copy/paste behaviour [36].

b) Debugging Tools for Program Understanding: Some developers tend to print pieces of text (*e.g.,*`System.out.print()` in Java) to locate faulty program elements but debugging tools are essential in any programming environment [37]. They help developers understand the dynamic behaviour of programs.

Hipikat [22], [23] is an Eclipse plug-in that for a project by analysing links between stored artefacts and recommending relevant ones. Also, Hipikat supports developers' foraging by reducing the cost of navigation among code and non-code artefacts, such as bug reports, CVS logs, and e-mails [23].

Romero *et al.* [38] extended the work by Katz and Anderson [39] 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.

JIVE [40], [41] is an Eclipse plug-in to analyse Java program executions, providing two kinds of runtime visualisations of Java programs - object diagrams and sequence diagrams.

DebugAdvisor [24] is a recommender system to improve debugging productivity by automating the search for similar issues from the past.

[9] studied the difficulties faced by developers when debugging in modern IDEs. They reported that the nature of the IDE affects the time spent by developers during debugging activities.

c) Automated debugging tools: Automated debugging tools require both successful and failed runs and do not support programs with interactive inputs [42]. Consequently, they have not been widely adopted in practice. Moreover, automated debugging approaches are often unable to indicate the "true" locations of faults [43]. 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 [44]. Researchers also found that automated debugging tools are rarely used in practice [44]. 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.* [45] 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 [42], [46] 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 [18] proposed *Omniscient debuggers*, and approach to support back in time navigation across previous program states. *Delta debugging* [47] 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 [48] proposed *object-centric debugging*, focusing on objects as the key abstraction execution for many tasks. Estler *et al.* [49] 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. Chen and Kim [50] proposed mining Stack Overflow QAs to leverage the large mass of crowd knowledge to aid developers while debugging programs. Salvaneschi and Mezini [51] presented RP Debugging to inspect and reason about the flow of changes through a reactive programming program.

VII. Conclusion

Debugging is a complex activity, both tedious and time consuming. During debugging, developers explore the source code of their systems, walking through different paths to locate faults. Debugging activities, thus, developers produce a lot of knowledge about systems. This information is however lost after the end of the developers' debugging activities. To prevent this information loss and allow developers to leverage knowledge of others' debugging activities during a new debugging activity, we introduce the concept of swarm debugging and the Swarm Debugging approach (SD).

SD uses developers' cooperative effort to capture and share knowledge, collecting information that are usually discarded in traditional debugging tools. The acquired knowledge is presented to developers using visualizations.

In this work, we conducted a case study to observe how developers toggle breakpoints. We collected debugging activities of 20 developers while they perform realistic debugging activities, to fix five real bugs in the Java open-source system JabRef, using our Swarm Debugging Infrastructure.

By analysing the statements on which the developers set breakpoints, we found that breakpoints are usually toggled on call statements (53% of breakpoints). Our results show that developers do not choose breakpoints simplistically; there is a **rational** behind each toggled breakpoint. We observed that different developers toggle the same places for the same task and different developers toggle the same type or method for different tasks. This finding has an important implication: there are locations (line of code, type, or methods) in the code that are prime choices for breakpoints. There is therefore an opportunity to recommend these locations to developers during new debugging sessions. In addition, we found in this paper that 53% (111/207) of breakpoints were toggled on ***call* statements** and only 1% (3/207) on *while-loop* statements.

These observations call for an investigation of whether breakpoints previously toggled by developers could help other developers. Consequently, we introduce the concept of co-breakpoints: developers who toggle breakpoints on a distinct type also toggle breakpoints in other types.

We conducted a qualitative study and a controlled experiment with professional developers to assess the effectiveness of Swarm Debugging. Results show that collecting and sharing debugging data is useful for bug location tasks. Developers can use Swarm Debugging Infrastructure to record their debugging data, allowing them to improve their debugging experience.

The promising results of our evaluation show the value of SD and call for more studies on software debugging. SDI is an open and freely available infrastructure that researchers can use to perform new empirical studies about debugging and–or software static and dynamic analysis. Developers can use SDI to understand IDE users' behaviours and requirements. This knowledge base is important to create new tools, using novel data mining techniques, to integrate different data sources. Educators can leverage SDI tools to teach interactive debugging techniques, tracing their students' debugging activities and evaluating their performance. Data collected by professional developers using SDI could also be used to educate students, *e.g.,* by showing them examples of good and bad debugging patterns.

Swarm Debugging brings actionable insights. First, software developers follow similar paths when debugging similar faults, hence researchers and tools vendors should explore recommendation systems to suggest paths to developers. Second, software developers follow certain patterns when navigating and toggling breakpoints during debugging, hence researchers and tools vendors should study, characterise, and use these patterns to recommend certain paths to developers. Moreover, following these two results, we also suggest that **debugging task could be divided into two activities**, one of **locating the faults**, which could benefit from the collective intelligence of other developers and could be performed by dedicated foragers, and one of **fixing the bugs**, which require deep understanding of the program and could be performed by dedicated builders. Hence, actionable results include recommender systems and a change of paradigm in the debugging of software programs.

Last but not least, the research community can leverage the SDI to conduct more studies to improve our understanding of developers' debugging activities, which could ultimately result into the development of whole new families of debugging tools that are more efficient and–or more adapted to the particularity of each debugging activity. Many open questions remain and this work is just a first step towards fully understanding how collective intelligence could improve debugging activities.

Our approach matches with a vision that IDEs should incorporate a general framework to capture and exploit IDE interactions, creating an ecosystem of developer-aware applications and plugins. Swarm Debugging is a first step towards intelligent IDEs, context-aware programs that monitor and reason about how their users interact with them, providing an environment to support the next generation of IDEs for crowd software engineering.

In future work, we plan to perform a large scale experiment on interactive debugging, collecting data from several systems and with different participants. We also intend to implement breakpoint recommendation systems as well as visualisations to support debugging, extending the Swarm Debug Infrastructure. Our ultimate goal is to improve the way developers debug and introduce the idea of crowd-sourcing to debugging.

Acknowledgments

We give our thanks to the developers who participated in our case study, supporting and improving our work with several insightful ideas and discussions. This work has been partly supported by the Natural Sciences and Engineering Research Council of Canada and the Canada Research Chair on Patterns in Mixed-language Systems.

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] P. Wainwright, "GNU DDD - Data Display Debugger," 2010.

[3] 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.

[4] 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

[5] 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.

[6] 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. 1–37, Sep. 2014.

[7] M.-a. Storey, L. Singer, B. Cleary, F. Figueira Filho, and A. Zagalsky, "The (R) Evolution of social media in software engineering," in *Proceedings of the on Future of Software Engineering - FOSE 2014*. New York, New York, USA: ACM Press, 2014, pp. 100–116.

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

[9] 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.

[10] T. Chow and D.-B. Cao, "A survey study of critical success factors in agile software projects," *Journal of Systems and Software*, vol. 81, no. 6, pp. 961–971, jun 2008. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2013.02.027$\backslash$nhttp://linkinghub.elsevier.com/retrieve/pii/S0164121207002208http://linkinghub.elsevier.com/retrieve/pii/S0164121207002208

[11] T. D. LaToza and A. van der Hoek, "Crowdsourcing in Software Engineering: Models, Motivations, and Challenges," *IEEE Software*, vol. 33, no. 1, pp. 74–80, jan 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7367992/

[12] S. Garnier, J. Gautrais, and G. Theraulaz, "The biological principles of swarm intelligence," *Swarm Intelligence*, vol. 1, no. 1, pp. 3–31, oct 2007. [Online]. Available: http://link.springer.com/article/10.1007/s11721-007-0004-yhttp://link.springer.com/10.1007/s11721-007-0004-y

[13] W. R. Tschinkel, "The architecture of subterranean ant nests: beauty and mystery underfoot," *Journal of Bioeconomics*, vol. 17, no. 3, pp. 271–291, oct 2015. [Online]. Available: "http://dx.doi.org/10.1007/s10818-015-9203-6http://link.springer.com/10.1007/s10818-015-9203-6

[14] T. Ball and S. Eick, "Software visualization in the large," *Computer*, vol. 29, no. 4, pp. 33–43, apr 1996. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=488299

[15] A. Cockburn, *Agile Software Development: The Cooperative Game, Second Edition*. Addison-Wesley Professional, 2006.

[16] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. D. Fleming, "How programmers debug, revisited: An information foraging theory perspective," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 197–215, 2013.

[17] F. Petrillo, Z. Soh, F. Khomh, M. Pimenta, C. Freitas, and Y.-G. Guéhéneuc, "On the effect of program exploration on maintenance tasks," in *Proceedings of the 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, August 2016.

[18] 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

[19] S. Demeyer, S. Ducasse, and M. Lanza, "A hybrid reverse engineering approach combining metrics and program visualisation," *Reverse Engineering, 1999. . . .*, no. Section 3, 1999. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=806958

[20] M. Fowler, "Datensparsamkeit," 2016. [Online]. Available: http://martinfowler.com/bliki/Datensparsamkeit.html

[21] 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.

[22] D. Cubranic and G. Murphy, "Hipikat: recommending pertinent software development artifacts," *25th International Conference on Software Engineering, 2003. Proceedings.*, no. Section 2, pp. 408–418, 2003.

[23] D. Čubranić, G. C. Murphy, J. Singer, and K. S. Booth, "Hipikat: A project memory for software development," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 446–465, 2005.

[24] 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.

[25] 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

[26] 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.

[27] J. Sillito, G. C. Murphy, and K. D. Volder, "Asking and answering questions during a programming change task," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 434–451, July/August 2008.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] A. Chi, O. Nierstrasz, and T. Gîrba, "Towards a moldable debugger," in *Proceedings of the 7th Workshop on Dynamic Languages and Applications - DYLA '13*, 2013, pp. 1–6. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2489798.2489801

[38] 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.

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

[40] P. Gestwicki and B. Jayaraman, "Methodology and architecture of JIVE," in *SoftVis '05 Proceedings of the 2005 ACM symposium on Software visualization*, vol. 1, no. 212, 2005, p. 95.

[41] J. K. Czyz and B. Jayaraman, "Declarative and visual debugging in Eclipse," in *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange - eclipse '07*, 2007, pp. 31–35. [Online]. Available: http://dl.acm.org.prox.lib.ncsu.edu/citation.cfm?id=1328279.1328286

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

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

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

[49] 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.

[50] F. Chen and S. Kim, "Crowd debugging," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*. New York, New York, USA: ACM Press, 2015, pp. 320–332. [Online]. Available: http://hunkim.cse.ust.hk/papers/chen-crowd-debugging-fse2015.pdfhttp://dl.acm.org/citation.cfm?doid=2786805.2786819

[51] G. Salvaneschi and M. Mezini, "Debugging for reactive programming," in *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*. New York, New York, USA: ACM Press, 2016, pp. 796–807. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2884781.2884815