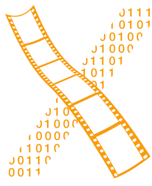

RWTH AACHEN UNIVERSITY

Debugging – But how? Unveiling Developer Behavior

Bachelor's Thesis
submitted to the
Media Computing Group
Prof. Dr. Jan Borchers
Computer Science Department
RWTH Aachen University

by
Lukas Liesenberg

Thesis advisor:
Prof. Dr. Jan Borchers

Second examiner:
Prof. Dr.-Ing. Ulrik Schroeder

Registration date: 27.11.2023
Submission date: 27.03.2024

Zentrales Prüfungsamt/Central Examination Office

RWTH AACHEN UNIVERSITY

Eidesstattliche Versicherung
Declaration of Academic Integrity

Liesenberg, Lukas
Name, Vorname/Last Name, First Name

395707
Matrikelnummer (freiwillige Angabe)
Student ID Number (optional)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel
I hereby declare under penalty of perjury that I have completed the present paper/bachelor's thesis/master's thesis* entitled

Debugging – But how? Unveiling Developer Behavior

selbstständig und ohne unzulässige fremde Hilfe (insbes. akademisches Ghostwriting) erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
independently and without unauthorized assistance from third parties (in particular academic ghostwriting). I have not used any other sources or aids than those indicated. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. I have not previously submitted this work, either in the same or a similar form to an examination body.

Aachen, 27.03.2024
Ort, Datum/City, Date

Unterschrift/Signature

*Nichtzutreffendes bitte streichen/Please delete as appropriate

Belehrung:
Official Notification:

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.
§ 156 StGB (German Criminal Code): False Unsworn Declarations
Whosoever before a public authority competent to administer unsworn declarations (including Declarations of Academic Integrity) falsely submits such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment for a term not exceeding three years or to a fine.
§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.
§ 161 StGB (German Criminal Code): False Unsworn Declarations Due to Negligence
(1) If an individual commits one of the offenses listed in §§ 154 to 156 due to negligence, they are liable to imprisonment for a term not exceeding one year or to a fine.
(2) The offender shall be exempt from liability if they correct their false testimony in time. The provisions of § 158 (2) and (3) shall apply accordingly.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:
I have read and understood the above official notification:

Aachen, 27.03.2024
Ort, Datum/City, Date

Unterschrift/Signature

Contents

Abstract xiii

Überblick xv

Acknowledgements xvii

Conventions xix

1 Introduction 1

2 Related Work 5

2.1 Code Navigation 5

2.2 IDE Debugging Tools and Techniques 7

2.3 Breakpoint Taxonomy 8

2.4 Debugging Behavior 9

3 Visual Studio Code Extension *DebugLog* **11**

3.1 Initial Considerations 11

3.2 Features 12

3.3 Technical Implementations 13
3.3.1 Accessing the Debugger 13
3.3.2 Detection of Print Statements and Log Statements in Source Code 16
3.3.3 Detecting Execution of Source Code Lines Containing Print/Log Statements 17

4 User Study 19

4.1 Study Design & Methodology 20
4.2 Study Procedure 22
4.3 Participants 23
4.4 Results & Evaluation 27
4.4.1 Quantitative Results 27
Previous Knowledge & Usage 28
Tool Difficulty 30
Tool Helpfulness 32
Confidence in Tool Usage 32
4.4.2 Qualitative Results 34
Usage Scenarios 34
Tool Opinions 35
4.5 Discussion 36
4.5.1 Conclusions 37

5 Summary & Future Work 39

5.1 Summary & Contributions 39

5.2 Future Work 40

A Informed Consent Form **41**

B User Study Questionnaires **43**

Bibliography **49**

Index **53**

List of Figures

2.1 Breakpoint Taxonomy 8

3.1 Debugger extension schematic 12

3.2 Example log 13

3.3 Standard configuration Sequence of the DAP 14

3.4 Python string syntax 15

3.5 Python import syntax 16

4.1 Guided example program 21

4.2 Age histogram 24

4.3 IDE experience histogram 24

4.4 Participants' rating of, satisfaction with, and confidence within their typical debugging process 26

4.5 Tool Usage Bar Chart 29

4.6 Tool Difficulty Bar Chart 30

4.7 Tool Helpfulness Bar Chart 31

4.8 Tool Confidence Bar Chart 33

A.1 Informed Consent Form 42

B.1 Demographics Page 1 44

B.2 Demographics Page 2 45

B.3 Demographics Page 3 46

B.4 Tool Overview 47

B.5 Tool Questionnaire 48

List of Tables

4.1 The IDEs of which our participants stated they are familiar with. 25

4.2 Wilcoxon Signed-Rank pairwise test results' p-values. Blue marked cells indicate $p \leq 0.05$. One redundant P column has been omitted. Abbreviations: **P**rint Statement, **B**reakpoint, **L**ogpoint, **C**onditional Breakpoint, **H**it Count Breakpoint 28

Abstract

Debugging software is considered the most time-consuming part of software development and software maintenance, often taking up about 35% of time. Debuggers have existed for over 40 years now and today, developers often utilize those that are built into Integrated Development Environments (IDEs). These IDE debuggers enable the developer to set Breakpoints, step through code, and, depending on the implementation, provide more advanced debugging features that speed up the process of debugging significantly. However, it has been shown that a large number of developers do not use the aforementioned features of IDE debuggers, but instead resort to debugging via Print Statements, i.e. placing them into lines before or after relevant source code parts to inspect variables or the execution progress in the console output. While many proposals for novel debugging tools have been made in the past, the causes for leaving behind advanced tools in favor of simple, but ineffective debugging methods like placing Print Statements, have been largely left unexplored.

In this thesis, we design and implement a Visual Studio Code extension *DebugLog* that logs all relevant debugging activities within a Python debugging session. We verify our design choices by conducting a user study, in which we inquire developers about how they debug and what they think about several debugging tools. To that end, we demonstrate each tool and let the participants rate them in terms of difficulty of usage, helpfulness, and confidence in usage via questionnaires. Based on the findings of the user study, we evaluate how our extension fits the needs of analyzing debugging behavior. We also evaluate and discuss the statements made by our participants to verify the findings in previous literature and to find an explanation for the preference for Print Statements.

Überblick

Das Debuggen von Software wird als der zeitaufwändigste Teil von Softwareentwicklung und Softwarewartung angesehen, häufig nimmt dieser 35% der Zeit ein. Debugger existieren nun schon seit über 40 Jahren und heutzutage nutzen Entwickler häufig ebenjene Debugger, welche in Integrated Development Environments (IDEs) integriert sind. Diese IDE Debugger ermöglichen es dem Entwickler, Breakpoints zu setzen, durch den Code zu schreiten und stellen, abhängig von der Implementierung, erweiterte Debuggingfunktionen zur Verfügung, welche den Prozess des Debuggens erheblich beschleunigen. Dennoch hat sich herausgestellt, dass eine große Zahl an Entwicklern die vorher erwähnten Funktionen nicht nutzen, sondern auf das Debuggen mithilfe des Platzierens von Ausgabebefehlen zurückgreifen, das heißt, jene Befehle in Zeilen zu setzen, welche sich vor oder nach relevanten Code-Zeilen befinden, um Variablen oder den Programmfortschritt auf der Kommandozeilenausgabe zu kontrollieren. Während es viele wissenschaftliche Vorschläge für neuartige Debugging-Werkzeuge in der Vergangenheit gab, sind die Gründe für das Außerachtlassen von erweiterten Debugging-Funktionen zugunsten einfacherer, aber ineffektiverer Methoden, wie zum Beispiel das Debuggen mithilfe des Platzieren von Ausgabebefehlen, weitgehend unerforscht geblieben.

In dieser Arbeit entwerfen und implementieren wir eine Visual-Studio-Code-Erweiterung *DebugLog*, welche alle relevanten Debugging-Aktivitäten in einer Python-Debugging-Sitzung aufzeichnet. Um unsere Entwurfsentscheidungen zu verifizieren, führen wir eine Studie durch, in welcher wir Entwickler über ihre Vorgehensweisen beim Debuggen und ihre Meinung bezüglich einiger Debugging-Werkzeuge befragen. Zu diesem Zweck demonstrieren wir jedes Werkzeug und lassen diese von den Teilnehmern in Bezug auf Schwierigkeit und Konfidenz in der Benutzung, sowie auf Nützlichkeit mithilfe von Fragebögen bewerten. Basierend auf den Ergebnissen der Studie evaluieren wir, wie gut unsere Erweiterung auf die Bedürfnisse des Analysierens von Debugging-Verhalten passt. Wir bewerten und diskutieren außerdem die Aussagen, die unsere Teilnehmer getätigt haben, um Erkenntnisse aus der Literatur zu verifizieren und eine Erklärung für die Präferenz für das Debuggen mithilfe von Ausgabebefehlen zu finden.

Acknowledgements

First of all, I would like to thank the Media Computing Group and Prof. Dr. Jan Borchers for giving me the opportunity to write this bachelor's thesis. Also, I would like to thank Prof. Dr. Ulrik Schroeder for taking the time to be my second examiner.

I would also like to express my appreciation towards my supervisor Adrian Wagner for taking much of his time advising me in realizing my thesis and being constantly available to answer my questions and giving me feedback.

Additionally, would like to thank each participant in the user study for their time spent and for their feedback to make this bachelor's thesis possible.

Lastly, I want to thank my family and friends for supporting me during the last few months.

Conventions

Throughout this thesis, we use the following conventions.

Text conventions

Definitions of technical terms or short excursuses are set off in colored boxes.

EXCURSUS:
Excursus are detailed discussions of a particular point in a book, usually in an appendix, or digressions in a written text.

Definition: *Excursus*

Source code and implementation symbols are written in typewriter-style text.

`myClass`

IDE debugging tools are written in Title Case.

The whole thesis is written in American English.

Chapter 1

Introduction

"Debugging is the dirty little secret of computer science."

—Lieberman [1997]

Debugging is an integral part of software development. The problem statement may seem simple: Identify and fix faults in a computer program. In reality, debugging is often considered the most time-consuming, but inevitable part of developing software and takes up more time than creating the original software in the first place [Zhao et al., 2008, Kernighan and Plauger, 1978, Zeller, 2009], it often takes up 35% of all software maintenance activities [Harrison and Cook, 1990]. To fix a bug, a developer must gain a deeper understanding of the program itself to identify the part of the source code causing the problem [Oman et al., 1989]. To that end, Integrated Development Environments (IDEs) provide developers with debugging tools, e.g. Breakpoints, that halt the program execution on a desired line of code which enables developers to inspect the current program state.

Debugging is an inevitable, time-consuming part of software development.

LaToza and Myers's [2011], Karrer et al.'s [2011], and Krämer et al.'s [2012] proposals all utilized knowledge obtained from the Call Graph of the execution of a program, i.e. a visualized form of all possible execution paths. It has been shown that these approaches not only boosted their

In the past, there were many IDE feature proposals, but few were adapted into today's IDEs.

participants' efficiency in debugging but also helped them to get an understanding of the unknown code base. However, these approaches have rarely been introduced into IDEs.

Advanced debugging features can boost efficiency but are rarely used.

For years, empirical research on IDEs and debugging in general has been widely neglected [Siegmund et al., 2014], although IDEs are extensively used and give developers access to a wide range of tools like Breakpoints, Stack Traces, and Variable Viewers. It has been shown that these features help developers considerably with debugging [Robillard et al., 2004], but the knowledge and use of advanced debugging tools remain low among developers [Beller et al., 2018]. Because of this, it is critical to analyze the behavior of developers not using these tools to improve debugging efficiency and speed up software maintenance and development. This thesis aims to find the root causes of this gap in helpfulness and usage.

To our knowledge, there is a gap of 15 years in research on debugging behavior in which no general-purpose study has been conducted.

Many developers debug using Print Statements, but we lack an explanation on when exactly they utilize them.

To address this issue, empirical research about the debugging behavior is needed. Gould and Drongowski [1974] investigated the debugging behavior of 30 participants using FORTRAN code printed out on several sheets of paper. For 15 years, a study done by Eisenstadt [1997] was the last general-purpose user study on debugging behavior [Siegmund et al., 2014]. In 2018 Beller et al. [2018] conducted a large-scale empirical study to find out what developers know about debugging and how they debug. To that end, they conducted an online survey and collected data with their *WatchDog* online IDE extension where developers could register voluntarily. Finally, they interviewed developers some of whom were actively developing debugging tools. Their findings indicated that most of the participants did not even start a debugging session and when they did, they often did not use advanced debugging tools built into the IDE. 61% of the participants of their online survey responded positively to the statement *"The best invention of debugging was printf debugging."*. However, they excluded measurements of when their participants placed Print Statements from their data collection. But exactly this knowledge can give explanations as to why developers are choosing to resort to Print Statements.

To achieve this, this thesis provides the Visual Studio Code[1] (VSCode) extension *DebugLog* to log a developer's navigation and debugging behavior. It detects all relevant usage of debugging tools built into the debugger of Visual Studio Code's Python extension and logs it to a log file. We evaluate our extension design choices with a user study. The goal of the user study is also to provide us with answers to the questions as to when, how, and why exactly developers use which IDE debugging tools.

DebugLog logs all debugging activities of a Python VSCode session.

The following chapters will give an overview of the relevant related work followed by a detailed description of the Visual Studio Code extension, then go into the details of the conducted user study and lastly, we will analyze our obtained results.

In Chapter 2 we will summarize the related work on debugging, as well as on code navigation, as these topics do overlap.

Chapter 3 will go into detail about our Visual Studio Code extension *DebugLog*, explaining the interaction with Visual Studio Code's Debug Adapter Protocol and the obstacles during the development of the extension to detect various debugging actions made by the developer.

To evaluate the choices made by us while designing the extension, Chapter 4 will detail the goal, the methodology, and the procedure of the user study that we conducted. Here, we will also analyze and discuss the results that we received from this study.

In Chapter 5, we will summarize what our thesis contributes to this field of research as well as give an outlook for future work.

[1] https://code.visualstudio.com (accessed 03.2024)

Chapter 2

Related Work

When we analyze the debugging behavior of developers, we need to consider several related research fields. Debugging is closely related and significantly influenced by considering the navigation through code and the usage of debugging tools.

2.1 Code Navigation

During debugging, developers often need to navigate source code for large amounts of time [Robillard et al., 2004]. Building a mental model of the source code based on information retrieved via navigation is critical for efficient debugging because developers need to relate an error to a malfunctioning point in the source code [Oman et al., 1989, Siegmund et al., 2014, Beller et al., 2018]. A study conducted by Ko et al. [2006] indicated that about 25% of a developer's time is spent on navigating source code.

With *Stacksplorer*, Karrer et al. [2011] proposed an IDE plug-in for Xcode[1] that visualizes the Call Graph of a given source code. It provides the functionality to navigate back and forth on the Call Graph to investigate different frames. The results of their user study and the answers given in

[1] https://developer.apple.com/xcode/ (accessed 03.2024)

the post-session questionnaire indicated that the participants appreciated *Stacksplorer's* benefit of understanding the source code more easily.

A proposal by Bragdon et al. [2010] discussed using the spatial arrangement of classes and methods by providing *Code Bubbles*. Classes and their containing methods are grouped visually into bubbles and the developer can open new, or extend, bubbles by either utilizing a search function or by clicking on the method names. Their user study results showed that the navigation time decreased significantly and the task completion times increased. DeLine et al. [2006] proposed a similar concept called *Code Thumbnails* by displaying a minified outline of the source code of one file on the side of the IDE. This concept was adapted as a built-in feature of some IDEs, e.g. in VSCode as Minimap[2].

By analyzing the previous navigation behavior of a developer, *Mylar* by Kersten and Murphy [2005] extends Eclipse[3] by an own file explorer, problems list, an *Outline* filter view, and a navigator view. They all utilize a degree-of-interest model to filter the displayed elements. For example, the file explorer shows the most interesting entries with a red background, and the *Outline* view displays related methods and automatically folds or unfolds methods viewed in the editor that are most relevant to the current edit context. Usually, the number of entries in the file explorer grows with the size of the project, so it becomes more crowded and impacts navigation time negatively. Participants of their study used the *Mylar* features significantly more often than the standard Eclipse features and opined that the modified file explorer was the best of their contributions. *NavTracks*, proposed by Singer et al. [2005], also introduced a modified file explorer, although it was based on previous navigational data by team members on a collaborative code project. The main research focus here lies on the onboarding of new team members and how to introduce them to a new code base efficiently.Čubranić and Murphy [2003] as well provided a tool called *Hipikat* that helped new devel-

[2] https://code.visualstudio.com/docs/getstarted/userinterface#_minimap (accessed 03.2024)

[3] https://eclipseide.org/ (accessed 03.2024)

opers who are joining a development team quickly understand and navigate the existing code base.

2.2 IDE Debugging Tools and Techniques

Usually, when referring to *debuggers*, we talk about *symbolic debuggers*, i.e. programs, that enable developers to utilize a set of tools, such as different types of Breakpoints that can be placed on lines of code by the developer. Once a program halt is triggered by a set Breakpoint, the developer can inspect or watch memory entities like variables or the call stack. The developer can often step through the source code line by line. One common debugger that supports these functions is the GNU Project Debugger (GDB) [Stallman et al., 1988].

Several research proposals exist that aim at extending the debugging toolset to enable more efficient debugging for developers. As mentioned previously, *Stacksplorer* [Karrer et al., 2011] and *Mylar* [Kersten and Murphy, 2005], apart from their contributions to code navigation, also provide unique tools that can be used for debugging purposes.

Whyline [Ko and Myers, 2008] lets the developer ask *Why?* and *Why not?* questions about specific results of program executions. It displays the causal chain of events that lead to the result in an interactive and explorable execution chain. Results showed that participants who used Whyline were twice as fast as the control group who used an unmodified version of Eclipse.

The concept of *time-travel debugging* has been explored by Buhse et al. [2019] with their tool *VeDebug*. They provide a video-like interface that enables the developer to rewind or fast-forward the current execution. The tool also provides *divergence points* that break the execution whenever the execution differs from a previous execution. Developers can skip to certain execution points and step through the code from there. However, an empirical user study was not conducted to evaluate the proposal.

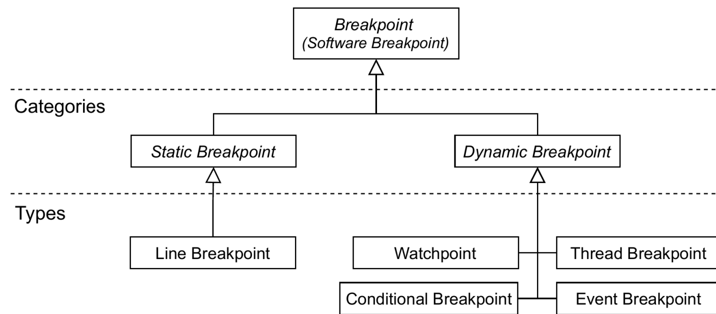

Figure 2.1: The Breakpoint taxonomy [Fontana and Petrillo, 2021].

2.3 Breakpoint Taxonomy

Fontana and Petrillo [2021] provided a taxonomy based on researching the Breakpoint types found in current IDEs and inquiring their participants about their own experiences and opinions (see Figure 2.1). Based on the results and the agreed-upon terminology and features of different Breakpoint types, they provided definitions of Breakpoints, which the following we are going to use in this thesis:

Definition:
Line Breakpoint

LINE BREAKPOINT:
Line Breakpoint is the breakpoint type associated with a line of code. *Where:* It is usually inserted over the line of code or eventually over a code tuple contained in the line of code, or else written directly in the code as a code stop instruction. *Trigger:* It is usually triggered when the line of code is hit during the system execution. *Effect:* When triggered, it causes the system execution interruption [Fontana and Petrillo, 2021].

CONDITIONAL BREAKPOINT:
Conditional Breakpoint is a breakpoint type associated with an expression. *Where:* It is usually inserted as a specific breakpoint type over a line of code, or it could be associated with other breakpoint types such as a configuration usually called property or action. *Trigger:* It is triggered whenever the condition related to its associated expression is true. This condition could be related to data, a count or the existence of a specific object. *Effect:* When triggered, it causes the system execution interruption [Fontana and Petrillo, 2021].

Definition: *Conditional Breakpoint*

TRACEPOINT:
Tracepoint logs a message on console output when execution through it. *Where:* It is usually inserted over the line of code. *Trigger:* It is usually triggered when the line of code with which it is associated is the next line to be executed by the system. *Effect:* When triggered, it displays a previously defined debug message on the console, **not interrupting the system execution**. *Complementary:* Furthermore, it could contain associated expressions that determines its activation only when the condition is true [Fontana and Petrillo, 2021].

Definition: *Tracepoint*

However, we will be calling the Line Breakpoint just *Breakpoint* and the Tracepoint *Logpoint* to adhere to the terminology used by VSCode. Furthermore, we also introduce the terminology of *Hit Count Breakpoint* to describe a Conditional Breakpoint whose condition is a count.

2.4 Debugging Behavior

The contributions of Beller et al. [2018] give interesting insights in developing behavior. As mentioned in the introduction, they conducted an online survey and collected debugging data via a voluntary online VSCode extension participation. Furthermore, they interviewed professionals to discuss their findings. The survey included questions about the participant's experience, favorite IDE, and how

they use the IDE debugging tools. A significant part of the contribution was also about what kind of debugging tools the participants already knew before. The survey was completed by 176 participants, of which 90.3% had at least three, and 36.9% had at least ten years of experience in software development. The results of the survey showed that 90% of all participants know and use standard Line Breakpoints, but are not familiar with more advanced debugging tools. 57% stated that they specify conditions on Breakpoints, and significantly fewer participants were familiar with specifying hit counts on Breakpoints. The last part of their survey consisted of a non-mandatory, open question asking about the participant's opinion on the statement "The best invention in debugging was printf debugging" based on a quote made by Kernighan [1978]. 61% of the participants responded positively to this statement. The IDE resulted in the insight that only 28.8% of participants started a debugging session. If Breakpoints were used, mostly only their enablement was changed, not advanced options. Subsequent interviews revealed that most interviewees praised Print Statements as a universal tool. One interviewee said that specifying hit conditions on breakpoints was "fuzzy" and once one knows the condition, one almost automatically understands the problem. While the contributions made by the authors are interesting, in this thesis we will try to verify the insights contributed by Beller et al., but we also want to inquire exactly when and where developers utilize which debugging tool.

Chapter 3

Visual Studio Code Extension *DebugLog*

In this chapter, we will detail the VSCode logging extension. The extension runs in the background of a VSCode coding session and can be configured allowing different debugging events to be switched on and off. In the following sections, we first look at the general VSCode environment, in which extensions are implemented, before presenting the implemented features and going into the technical implementations.

3.1 Initial Considerations

We choose to develop the extension in VSCode due to its widespread adoption among developers[1]. The big extension ecosystem and its cross-platform capabilities are also an advantage. We also choose Python as our target programming language because it is popular, especially among developers learning to code[2].

[1] https://survey.stackoverflow.co/2023/#section-most-popular-technologies-integrated-development-environment (accessed 03.2024)

[2] https://survey.stackoverflow.co/2023/#section-most-popular-technologies-programming-scripting-and-markup-languages (accessed 03.2024)

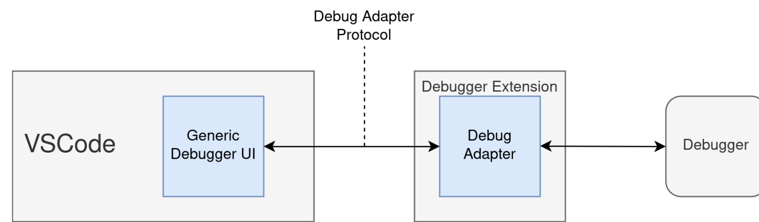

Figure 3.1: Schematic of a VSCode debugger extension. The debugger extension implements a debug adapter that ensures communication between VSCode and the installed debugger, in our case the Python runtime. Our extension will be listening to the communication via the Debug Adapter Protocol (DAP).

VSCode extensions are implemented by utilizing the VSCode Extension API[3] and are written in the TypeScript[4] programming language. Extensions within VSCode can change the IDE's visuals, add language support, implement debuggers, or add new commands or functions. Our extension however, will listen to the developer's inputs and actions within the user interface (UI) and their interactions with the debugger implemented by the Python extension[5]. To ensure functionality, *DebugLog* depends on the Python extension being installed.

3.2 Features

DebugLog can log a large variety of actions that are not only related to debugging but also include more common actions such as changing the currently viewed code file or editing it. Additionally, the user can exclude certain events from logging via the VSCode settings. The logging format is illustrated in Figure 3.2.

We also implement a *highlight Print/Log Statements* command that can be executed via the built-in command

[3]https://code.visualstudio.com/api (accessed 03.2024)
[4]https://typescriptlang.org (accessed 03.2024)
[5]https://marketplace.visualstudio.com/items?itemName=ms-python.python (accessed 03.2024)

```
TIMESTAMP - EVENT :: INFO :: LOCATION :: RAWDATA
    └── Date.now()
                └── ADD_BREAKPOINT
                        └── (true, 'flag === "stop"', '', '', 42)
              ./main.py@12 ──┘
                                  { ... } ──┘
```

Figure 3.2: Logging format with example. It denotes that *now*, the debugger has detected that a *Breakpoint* was *added* on line *12* in *main.py*, only breaking when the string variable `flag` equals `"stop"`. It has no hit count and no Logpoint logging message. This tells us this must be a *Conditional Breakpoint*. The internal ID is *42*.

palette. It selects all Print/Log Statements in the current file.

3.3 Technical Implementations

To detect the developer's actions, we use event listeners provided by the VSCode Extension API. Furthermore, we search source files for Print and Log Statements via regex expressions. The next sections will present our implementations regarding these events and the detection of Print Statements and Log Statements.

3.3.1 Accessing the Debugger

As shown in Figure 3.1, we need to listen to the communication between VSCode and the debug adapter to intercept and interpret debugging events. This communication is done via the Debug Adapter Protocol (DAP). Figure 3.3 shows a typical debugging initialization. To listen to the events and requests of the debugging ses-

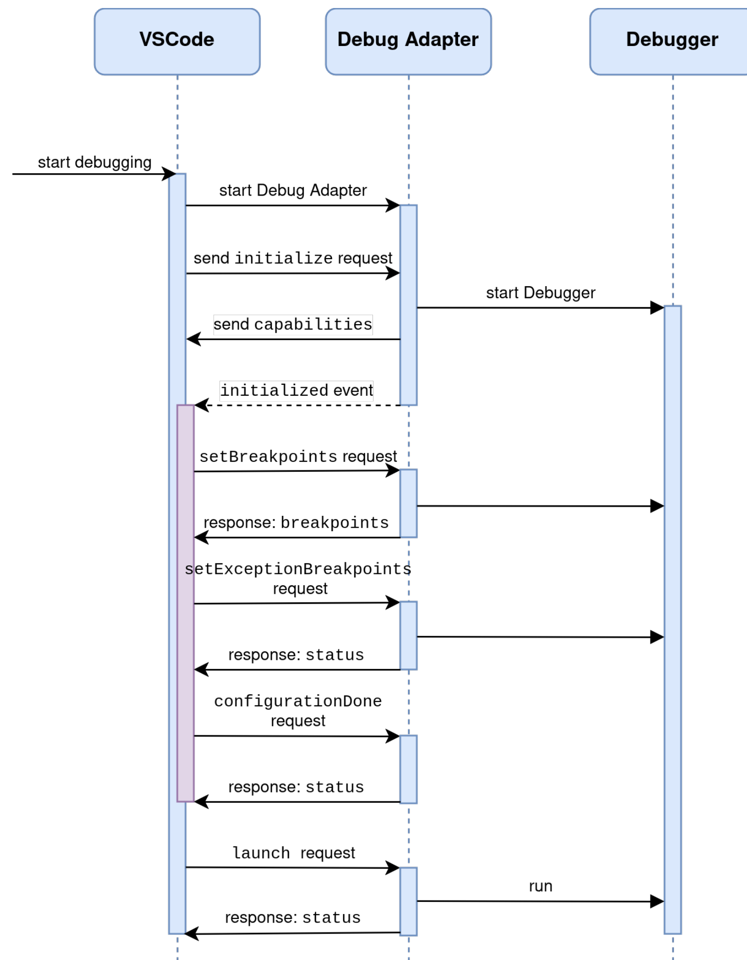

Figure 3.3: Sequence diagram of a standard configuration process of the DAP. Every debugging session gets initialized and then configured by setting Breakpoints and Exception Breakpoints if any exist. All relevant events are forwarded to the debugger. After the configuration is done, the session is launched depending on the launch configuration.

sion, the VSCode API provides the registration of a `DebugAdapterTrackerFactory` that we are going to implement in an own class file.

The `DebugAdapterTracker` provides two event listeners: `onWillReceiveMessage` and `onDidSendMessage`.

```
doubleQuote = f" 'test' \
    foo"

singleQuote = f' "test" \
    foo'

tripleQuote1 = """ 'test'
    foo"""

tripleQuote2 = ''' "test"
    foo'''

# this is a comment
```

```
stringRegex = /f{0,1}(?:"(?:[^"\\]|\\.)*"|'(?:[^'\\]|\\.)*')/gs

stringWithTripleQuoteRegex = /(?:'''|""")(.*?)(?:'''|""")/gs

commentRegex = /#.*$/gm
```

Figure 3.4: Top: Different types of strings supported by the Python syntax: f-strings, multiline strings, different types of quotation marks, quotation marks mixed within strings, and line continuations with backslashes, also a comment. **Bottom:** Regular expressions implemented in TypeScript for the cases above.

We can use these listeners to get notified whenever the debug adapter sends or receives DAP messages illustrated in Figure 3.3 and handle the message contents as desired. Whenever the execution halts on a Breakpoint, a `stopped` event is sent to the debugger. It has a `reason` attribute which can be for example `breakpoint`, `step`, or `exception`. If the `reason` attribute equals `breakpoint`, we log a `HIT_BREAKPOINT` event. After each `stopped` event, a `stackFrames` request is sent to the debug adapter, which is answered with the according active frame containing our current line of execution. Based on this information, we can derive which Breakpoint was hit and enrich our `HIT_BREAKPOINT` log event with additional information about log messages, hit counts, or conditions. Additionally, whenever the debug adapter sends a `setBreakpoints` response, we update our internal Breakpoints array if needed and log a `ADD_BREAKPOINT` event for every new Breakpoint.

```
import logging as foo

from logging import *

import logging

from logging import warn

from logging import (debug,
                     warn,
                     log)

from logging import warn,
                    debug \
                    log as foo
```

```
libraryAliasRegex = /import (?:.*, +)*logging(?: as (?:([^,\n]+),.+$|(.*)
    $))/gm

libraryAsteriskRegex = /from logging import \*/g

libraryStandardRegex = /import logging(?: *$|,.*$)/gm

methodImportRegex = /from logging import ([^(\\]+)$/gm

methodImportBracesRegex = /from logging import ([^(\\]+)$/gm

methodImportBackslashesRegex = /from logging import ((?:[^(]+, *\\\n *[^\
    n]+))/gs
```

Figure 3.5: Top: Different types of logging import statements supported by the Python syntax: Whole library, method selections and aliases. **Bottom:** Regular expressions implemented in TypeScript for the cases above.

3.3.2 Detection of Print Statements and Log Statements in Source Code

In addition to debug events, we also want to send a log message whenever the user adds, removes, or modifies Print Statements. We extended this idea to also include Log Statements, i.e. invocations of the standard TypeScript logging library.

To that end, for each file in our workspace, we first need to ignore all text ranges that are included within comments or strings. We solve this via regular expressions. Python supports a multitude of different notations for strings (see

Figure 3.4 (Top)). We match them using the regular expressions presented in Figure 3.4 (Bottom) and store them in an array `ignoredRanges` of type `Range[]` which is a type provided by the VSCode API to describe ranges in code files.

Next, we collect all possible notations of Python logging invocations. To use logging in Python, the logging library must be imported first. There are multiple ways to import libraries in Python, as shown in Figure 3.5 (Top). If only certain methods are imported, we only have to scan for these methods. Furthermore, if aliases were defined, these also need to be considered. We collect all matched notations into a `possibleLogsRegex` regular expression that simply concatenates all notations with an *OR* operation and search for all occurrences. Finally, after discarding those that are within any of the ranges in `ignoredRanges`, we successfully collected all Log Statements.

Finally, we search for Print Statements using the simple regular expression `/print\(/g` that returns us all beginning parts of Print Statements up until the opening brace. Because Print Statements can include other method invocations that use braces, we additionally search for the last closing brace such that there is an even amount of opening and closing braces not contained within `ignoredRanges` between this brace and our `print(` part. This way, we not only consider Print Statements containing pure strings but also Print Statements that contain other methods returning printable objects. For example, we can now detect invocations of the form `print("foo " + buildString())`.

3.3.3 Detecting Execution of Source Code Lines Containing Print/Log Statements

In the previous section, we outlined the implementation of the detection of Print Statements and Log Statements. However, we also want to find out when the debugger processes the statement because this can give us information about the execution sequence of the program while debugging. The debug adapter does not give us informa-

tion about the specific execution progress of the debugger. Therefore, without rewriting the Python debugger itself to expose this information, we are utilizing Logpoints. Whenever the debug adapter receives an `setBreakpoints` event from DAP, we copy that request information and attach additional Logpoints to the `breakpoints` array. The location of these Logpoints corresponds to lines on which the user has placed a Print Statement or a Log Statement. We fill the log message of the Logpoint with a "DebugLog" marker, information about the location with path information to the source code file, and its arguments, i.e. expressions between the invocation braces. This way, the Logpoints are not displayed in the VSCode UI and remain invisible to the developer, because the UI only retains visual indications of Breakpoints that are set within itself. In the `logActivity` function, we distinguish between normal `output` events and those that contain the "DebugLog" marker and log the information in the log message. In the case that there are multiple Print/Log Statements in a code line, we insert a delimiter in the log message and concatenate both messages.

Chapter 4

User Study

The user study aims to determine what developers think about certain tools used while debugging with an IDE and in what situations they prefer using them. To that end, we want to answer the following research questions:

RQ1: What debugging tools are already known by developers?

RQ2: In which situations are they used within an IDE?

RQ3: How helpful are specific tools?

RQ4: How satisfied are developers with the solution gained by using a specific tool?

In the following, means will be denoted by $\bar{x}$, medians by $\tilde{x}$, and standard derivations by σ. Results of means, medians, or standard derivations are rounded to the second digit. Results of empirical tests such as p- or χ^2-values are rounded to the sixth digit.

4.1 Study Design & Methodology

To evaluate our participant's previous knowledge in IDE debugging as well as basic demographics, we ask them to fill out a questionnaire (see Appendix B). We collect information about age, gender, profession, and the participant's knowledge about IDEs as well as which specific IDEs they have experience in. Furthermore, we also inquire about our participants' experience in Python and IDE debugging in general.

Concerning debugging, we also check the participant's familiarity with debugging tools that are integrated into IDEs. Also, the participants are asked to rate their typical debugging process in terms of satisfaction and confidence, because we want to determine whether there is a difference between the usage of tools and the developer's rating of the outcome.

In the second part of the user study, we guide the participants through a small Python code example prepared within the VSCode IDE (see Figure 4.1). We ask the participants to use specific tools in specific source code lines. While the participant follows our instructions using one tool, we explain the functionality of the tool and how its features help in debugging. We select the following tools for this part:

1. Print Statement
2. Breakpoint
3. Logpoint
4. Conditional Breakpoint
5. Hit Count Breakpoint

We pick these debugging tools based on literature related to IDE debugging. As stated by Beller et al. [2018], debugging via the usage of Print Statements is praised as a universal tool by their interviewees. Also, in their online debugging survey, 71.6% of participants stated that they use

```python
1  def is_number(string):
2      return string.isnumeric()
3
4  def add_two_numbers(firstNumber, secondNumber):
5      result = firstNumber - secondNumber
6      return result
7
8  firstUserInput = input("Please the first number: ")
9
10 if not is_number(firstUserInput):
11     print("Please only enter numbers.")
12     exit()
13
14 secondUserInput = input("Please enter the second
       number: ")
15
16 if not is_number(secondUserInput):
17     print("Please only enter numbers.")
18     exit()
19
20 result = add_two_numbers(int(firstUserInput), int(
       secondUserInput))
21
22 print("Your calculation was: " + firstUserInput + "+
       " + secondUserInput)
23 print("Your result was: " + str(result))
```

Figure 4.1: Python code example used in the user study. The participants were instructed to use Print Statements, Breakpoints, Logpoints, Conditional Breakpoints, and Hit Count Breakpoints at specific lines of code as a way of introducing the functionality of each tool (see Figure B.4)

Print Statements. To verify this, we include them in the second phase. We choose the IDE debugging tools based on Fontana and Petrillo's [2021] Breakpoint Taxonomy (see Figure 2.1). While standard Breakpoints fall under the *Static Breakpoint* category, Conditional Breakpoints and Hit Count Breakpoints are counted as *Conditional Breakpoints* while standard Breakpoints are included in the *Line Breakpoint* type. Furthermore, regarding the third level of the taxonomy, both fall under the *Conditional Breakpoint* type. However, Logpoints are excluded from the taxonomy because they do not cause an execution stop. But because six out of nine IDE's that were investigated by Fontana and Petrillo [2021] include the functionality similar to a Logpoint and Logpoints were not investigated by Beller et al.

[2018], we choose to include them in our study to gain insights into the usage of Logpoints among developers.

To get insights into the participant's previous knowledge and opinion of the introduced tools, after each demonstration, we conduct a short interview about the shown tool. The investigator asks questions about whether the participant already knew the tool (RQ1) and if yes, in what situations they usually implement it (RQ2). We also inquire about the frequency of their usage while debugging.

To find out how satisfied the participants were with the tool's functionality, we ask them to rate how easy to use and how helpful they think the current tool is. We also want to know how they rate their confidence in using the tool. (RQ3 & RQ4) After these questions, the investigator asks additional questions about their choices, asking them to elaborate.

4.2 Study Procedure

After being seated, the participant was asked to read and fill out the Informed Consent Form (see Appendix A). The study took place in a quiet room and the participants were provided with water and were free to grab a variety of snacks. For the second part of the study, the participants were free to choose the mouse's scrolling direction and the keyboard layout. This phase was conducted on a Macbook Pro 16" 2021 M1 Max. The standard built-in Mac American ANSI keyboard, a German Windows (QWERTZ) ISO USB keyboard, and a USB mouse were provided as options. If desired, the scroll direction and the keyboard and keyboard layout were changed by the investigator.

The study started with the demographics part being handed out to the participants (see Appendix B). Page 1 was handed out on its own, while pages 2 and 3 were handed out together. This was done deliberately. While page 1 included the question of what *features of IDEs* the participants already know, on page 2 we asked what *debugging tools integrated into IDEs* the participants already know.

This way, we can find out whether our participants consider debugging tools as a part of their debugging workflow and whether they consider them a part of their programming experience.

After filling out the demographics part, the investigator started recording the participant's screen and the computer's microphone. As stated in the Informed Consent Form, the recording was only used for later analysis purposes in the case that the investigator was not able to note statements made by the participant quickly enough.

We recorded the study for later analysis purposes.

Then, we moved on to the second part. After handing out the tool overview (see Figure B.4) the participant was guided through the debugging tools by debugging the prepared Python code example (see Figure 4.1) and received explanations. After each tool, the investigator interviewed the participant following the questionnaire sheet (see Figure B.5). Additionally, the investigator asked for the reasoning for certain choices to gain further insights into the decision-making. At the end of each tool questionnaire, the investigator also asked for any further opinions, remarks, or critiques related to the tool.

At the end of the study, the participants were asked for further opinions about debugging in general.

4.3 Participants

A total of 18 people participated in our user study. The participants were recruited via an online appointment scheduling tool from February 7th to February 13th, 2024. The only requirements for the participants were a basic knowledge of IDEs and Python. The ages ranged from 19 to 31 years (four female, thirteen male, one none, $\bar{x} = 25.44$, $\tilde{x} = 25.5$, $\sigma = 2.59$, see Figure 4.2). 17 of our participants had a Computer Science background. We recruited one person (P6) who studied Biotechnology.

18 participants, all but one had a Computer Science background.

Of the 17 participants with a Computer Science background, eleven were students (ten in Computer Science,

Eleven CS participants were students, the other six were employed in a CS-related job.

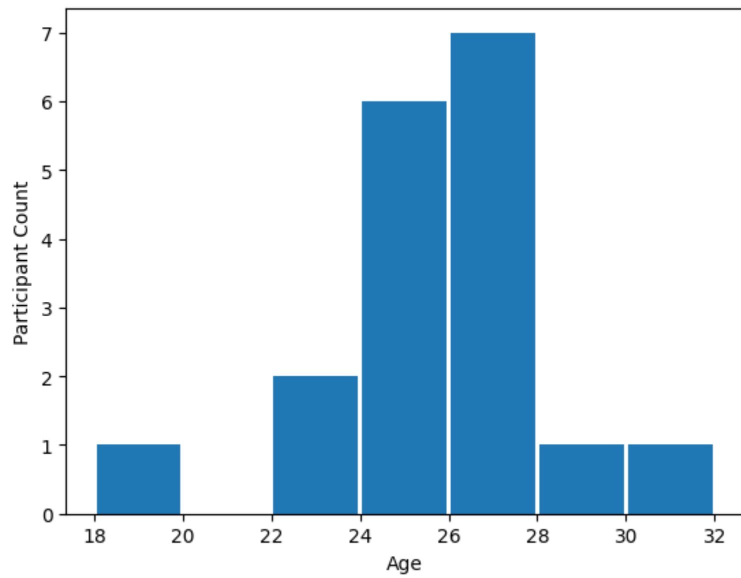

Figure 4.2: Age histogram of our participants, with $\bar{x} = 25.44$, $\tilde{x} = 25.5$, $\sigma = 2.59$.

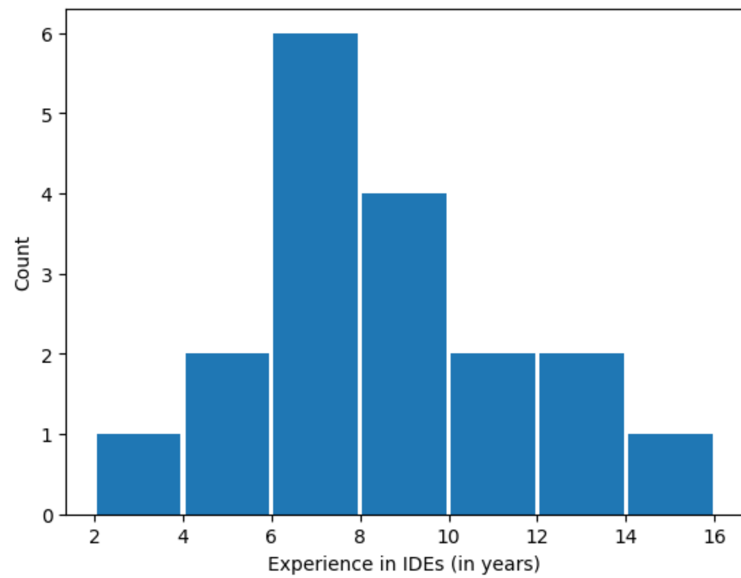

Figure 4.3: Histogram of the experience in IDEs of our participants, with $\bar{x} = 7.97$, $\tilde{x} = 7.5$, $\sigma = 3.11$.

one in Media Informatics). The remaining six worked as researchers (two), software developers (two), IT Security evaluators (one) and pen-testers (one). This group had spent an average of 2.08 years in their occupation.

IDE	Participant Count
Visual Studio Code	16
IntelliJ	11
Eclipse	6
PyCharm	6
Visual Studio	4
Android Studio	3
BlueJ	3
vim	3
Xcode	3
Emacs	2
Rider	2
Atmel Studio	1
Brackets	1
CLion	1
Code::Blocks	1
Delphi	1
LunarVim	1
Overleaf	1
Spyder	1
Unity	1
Unreal Engine	1

Table 4.1: The IDEs of which our participants stated they are familiar with.

All participants were familiar with VSCode.

All of our participants have used an IDE before. Their experience with IDEs ranged from three to 15 years ($\bar{x} = 7.98$, $\tilde{x} = 7.5$, $\sigma = 3.11$, see Figure 4.3). 16 of our participants had experience with VSCode, other major known IDEs were IntelliJ[1], Eclipse[2], and PyCharm[3] (see Table 4.1). All participants said that they have experience in Python development.

Regarding the participants' debugging process, we can ob-

[1]https://jetbrains.com/de-de/idea/ (accessed 03.2024)
[2]https://eclipseide.org/ (accessed 03.2024)
[3]https://jetbrains.com/de-de/pycharm/ (accessed 03.2024)

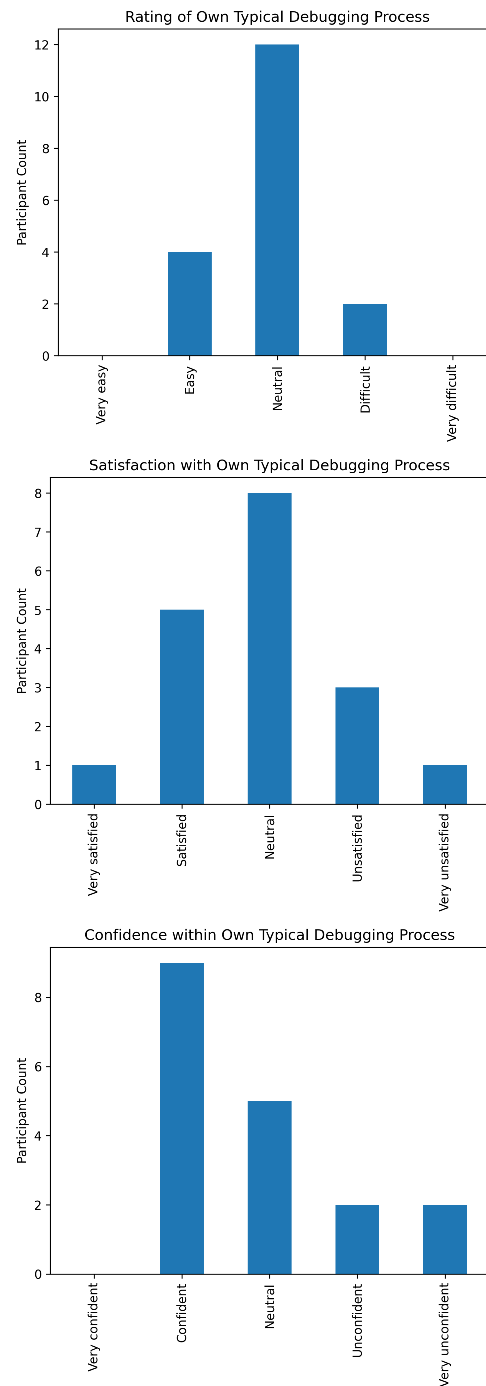

Figure 4.4: Distribution of our participants' assessment of their debugging process in regards to rating, satisfaction, and confidence.

serve in Figure 4.4 that the general rating given by the participants to their debugging process is generally *Neutral* with twelve participants. Four participants chose the option *Easy* while two chose *Difficult*. No participant chose to answer *Very easy* or *Very difficult*. Concerning satisfaction, the answers show that options were chosen more diversely. While *Neutral* is still the most chosen option with 8 votes, we also have one chosen option each for *Very satisfied* and *Very unsatisfied*. Five participants rated their satisfaction with *Satisfied*, while three participants chose *Unsatisfied*. Regarding confidence, no participant chose to answer *Very confident*. *Confident* is the most answered option with nine participants, followed by *Neutral* with five participants and *Unconfident* and *Very unconfident* with two participants each.

4.4 Results & Evaluation

In this section, we present and analyze the quantitative and qualitative results of our user study to address our research questions. Our goal is to find out to what extent they can be answered by the results. After that, we will discuss them in the next section.

4.4.1 Quantitative Results

For the analysis of the results of our three five-point Likert scales (see Figure B.5), we will use a Friedman test for each scale since our data turned out to be not normally distributed. To that end, we convert our Likert scales into discrete point scales ranging from one to five with the most negative option corresponding to *1* and the most positive option corresponding to *5*. Furthermore, we will also utilize Wilcoxon signed-rank tests as post-hoc pairwise testing to find potential significant differences in the rating of the different tools. For these tests, we will test against a significance threshold of $\alpha = 0.05$.

Difficulty

	B	L	C	H
P	0.003892	0.032509	0.011832	0.011699
B	-	0.527089	0.738883	0.617075
L	-	-	0.317311	0.355532
C	-	-	-	0.763025

Helpfulness

	B	L	C	H
P	0.013271	0.095581	0.056851	0.204357
B	-	0.08826	0.379537	0.004258
L	-	-	0.304937	0.007974
C	-	-	-	0.005064

Confidence

	B	L	C	H
P	0.011699	0.001766	0.010978	0.005043
B	-	0.812742	0.815803	0.873492
L	-	-	1	1
C	-	-	-	1

Table 4.2: Wilcoxon Signed-Rank pairwise test results' p-values. Blue marked cells indicate $p \leq 0.05$. One redundant P column has been omitted. Abbreviations: **P**rint Statement, **B**reakpoint, **L**ogpoint, **C**onditional Breakpoint, **H**it Count Breakpoint

Previous Knowledge & Usage

All participants were familiar with Print Statements and Breakpoints, other tools were less known.

Figure 4.5 shows the previous usage of our chosen debugging tools. Print Statements and Breakpoints are known by all participants. More advanced debugging tools like Logpoints, Conditional Breakpoints, and Hit Count Breakpoints are less known with only five participants who were familiar with Logpoints (27.7%), eleven with Conditional Breakpoints (61.1%), and five with Hit Count Breakpoints (27.7%).

Furthermore, regarding the usage of debugging tools, Figure 4.5 also shows that all participants who stated that they knew Print Statements had also used it before. For Logpoints and Conditional Breakpoints, three participants each stated that they did not use it, even though they knew of its

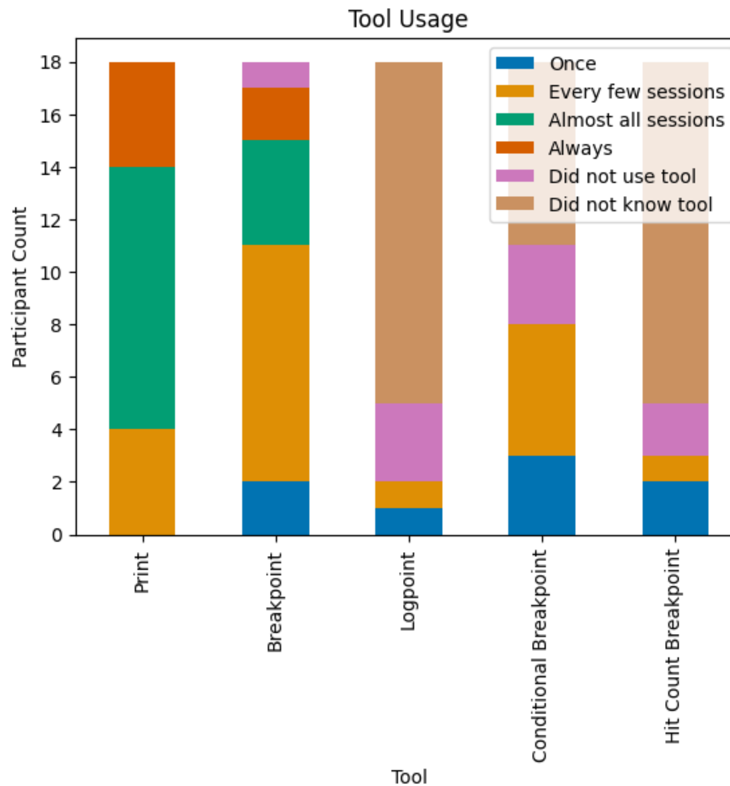

Figure 4.5: Distribution of our participants' usage of our chosen debugging tools.

existence. Hit Count Breakpoints featured two such participants, and normal Breakpoints only one. Furthermore, it showed that Print Statements and Breakpoints are the only tools that some participants use *Always* (Print Statements: four, Breakpoints: two). The option *Almost all sessions* was answered by ten participants for Print Statements and by four participants for Breakpoints. The option *Almost all sessions* was picked by no participant for the other tools as well. Instead, option *Once* was chosen by one, three, and two participants while option *Every few sessions* was chosen by one, five, and one participant for Logpoint, Conditional Breakpoint, and Hit Count Breakpoint respectively.

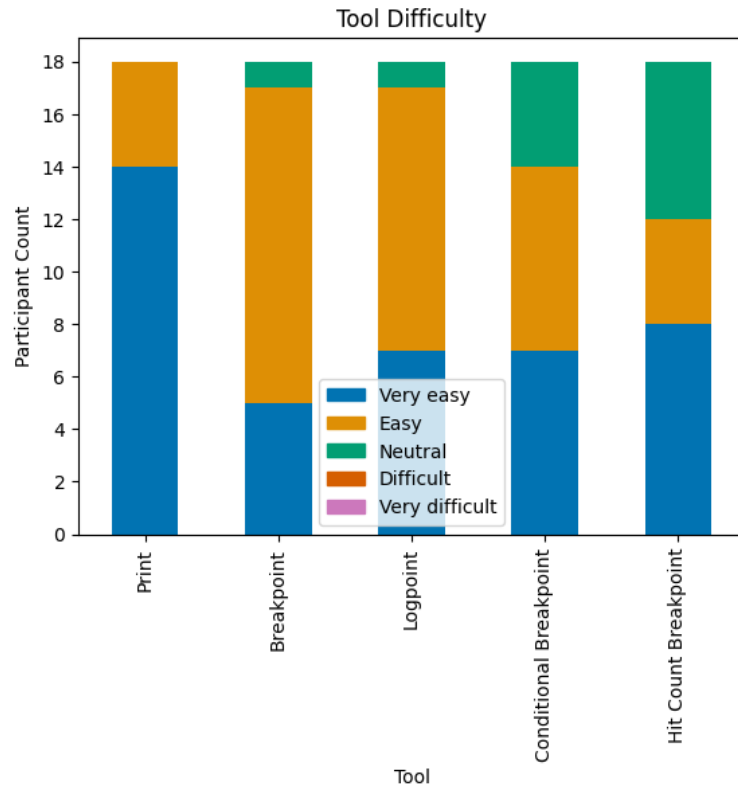

Figure 4.6: Distribution of our participants' assessed difficulty of different tools. Note that none of our participants chose to answer *Difficult* or *Very difficult*.

Tool Difficulty

Figure 4.6 shows the perceived difficulty of the tools. No participant chose the option *Difficult* or *Very difficult* for any of the tools. For Print Statements only the options *Very easy* and *Easy* were chosen with 14 participants for *Very easy* and four participants for *Easy* ($\bar{x} = 4.78$, $\tilde{x} = 5$, $\sigma = 0.45$). For all the other tools it can be noted that going from Breakpoint ($\bar{x} = 4.22$, $\tilde{x} = 4$, $\sigma = 0.55$) over Logpoint ($\bar{x} = 4.33$, $\tilde{x} = 4$, $\sigma = 0.59$) and Conditional Breakpoint ($\bar{x} = 4.17$, $\tilde{x} = 4$, $\sigma = 0.79$) to Hit Count Breakpoint ($\bar{x} = 4.11$, $\tilde{x} = 4$, $\sigma = 0.9$), the number of chosen *Easy* options is decreasing with twelve, ten, seven, and finally four participants respectively. Contrarily, the number of chosen *Very easy*

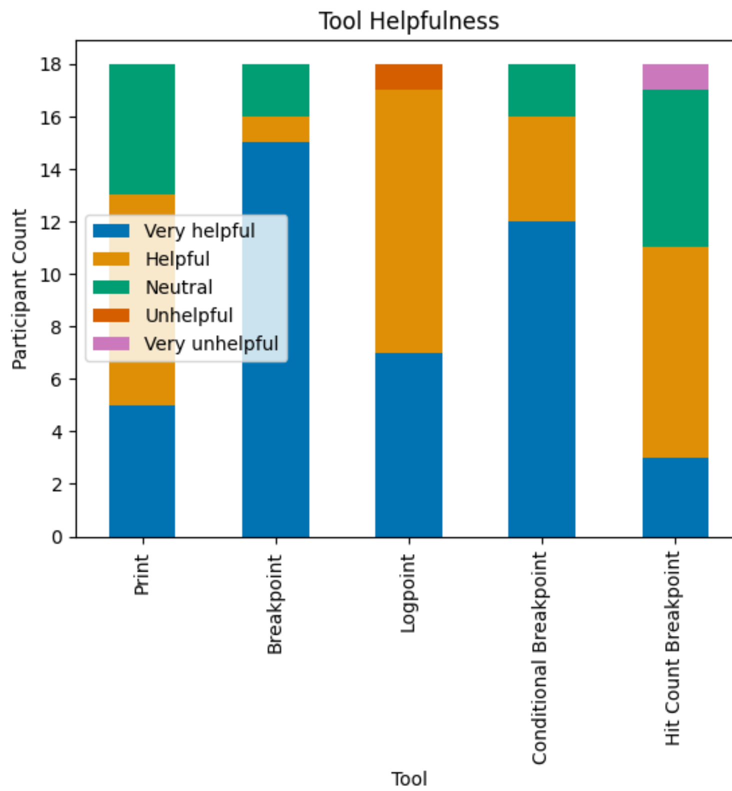

Figure 4.7: Distribution of our participants' assessed helpfulness of different tools.

and *Neutral* options is increasing, with five and one, seven and one, seven and four to eight and six respectively. A Friedman test indicated significant differences in participants' perceived difficulty among the tools ($\chi^2 = 33.655469$ and $p = 0.009298$). Wilcoxon tests showed that for Print Statements, the ratings all differ significantly. The greatest significance ($p = 0.003892$) was between Print Statements and Breakpoints. For all the other tools, the differences are largely insignificant, i.e. the participants did not feel that any tool was easier to use than another (see Table 4.2).

Tool Helpfulness

Figure 4.7 shows the tool helpfulness. No participant chose options worse than *Neutral* for all tools except for Logpoint and Hit Count Breakpoint which had one *Unhelpful* and one *Very unhelpful* respectively. Ranking the tools by the number of participants who gave them the rating *Helpful* or *Very helpful*, the Logpoint ($\bar{x} = 4.28$, $\tilde{x} = 4$, $\sigma = 0.75$) comes first with 17 participants agreeing on its helpfulness, followed by Breakpoint ($\bar{x} = 4.72$, $\tilde{x} = 5$, $\sigma = 0.67$) and Conditional Breakpoint ($\bar{x} = 4.56$, $\tilde{x} = 5$, $\sigma = 0.7$) with 16 participants each, Print Statement ($\bar{x} = 4$, $\tilde{x} = 4$, $\sigma = 0.77$) with 13, and Hit Count Breakpoint ($\bar{x} = 3.67$, $\tilde{x} = 4$, $\sigma = 0.97$) with eleven participants. Breakpoint had the most participants choosing the option *Very helpful* with 15 participants. A Friedman test showed significance ($\chi^2 = 27.647343$ and $p = 0.049231$). Investigating the results of Wilcoxon tests, we can observe significant differences between Print Statements and Breakpoints, as well as between Hit Count Breakpoints and all the other tools except Print Statements. Most of these p-values are even less than 0.01 (see Table 4.2).

Confidence in Tool Usage

Figure 4.8 shows the participants' assessed own confidence in the usage of the tools. Again, no participant chose the most negative option *Very unconfident*. Most of the participants (15) felt *Very confident* in using Print Statements ($\bar{x} = 4.78$, $\tilde{x} = 5$, $\sigma = 0.55$). The least confidence was with Breakpoints ($\bar{x} = 4.11$, $\tilde{x} = 4$, $\sigma = 0.9$) where only, but still a significant amount of twelve participants, felt *Confident* (four) or *Very confident* (eight) with using them. For Logpoints ($\bar{x} = 4.06$, $\tilde{x} = 4$, $\sigma = 0.73$), Conditional Breakpoints ($\bar{x} = 4.06$, $\tilde{x} = 4$, $\sigma = 0.87$), and Hit Count Breakpoints ($\bar{x} = 4.06$, $\tilde{x} = 4$, $\sigma = 0.8$), most participants (nine, eight, and seven, respectively) answered *Confident* with *Very confident* being the second-most answered option. One participant answered *Unconfident* regarding Conditional Breakpoints which was the only occurrence of this option in all tools. A Friedman test indicated a significance with

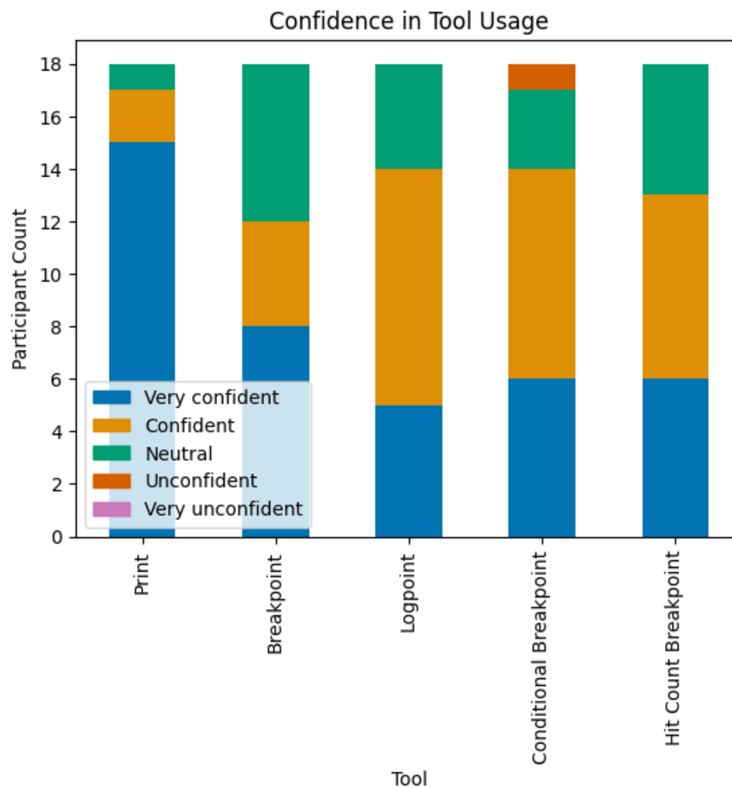

Figure 4.8: Distribution of our participants' assessed own confidence in tool usage. Note that none of our participants chose to answer *Very unconfident*.

$\chi^2 = 42.732167$ and $p = 0.000525$. Regarding the Wilcoxon test results, there are significant differences between Print Statements and all the other tools with the lowest p-value of 0.005043 resulting from the comparison between Print Statements and Hit Count Breakpoints. Wilcoxon tests for the three pairs Logpoints versus Conditional Breakpoints, Logpoints versus Hit Count Breakpoints, and Conditional Breakpoints versus Hit Count Breakpoints could not be conducted, because the options chosen by our participants were too similar resulting in ties. The remaining non-zero ranks did not suffice for a normal approximation. Therefore, we can not prove that there are significant differences between these pairs. In Table 4.2, these are denoted by *1*.

4.4.2 Qualitative Results

During the study, the participants were asked to state their usage scenarios for the tools as well as explanations for their ratings.

Usage Scenarios

Many of our participants stated that Print Statements are usually the first tool that they use when debugging. Ten of our participants stated that they use them to check whether certain functions were executed or to check the execution flow. Nine participants mentioned they use it to quickly check the value of variables.

Many of our participants said that they move forward to debugging using Breakpoints whenever Print Statements do not suffice anymore. In general, our participants stated that they use them more often when the code is more complex and they need to have the Step Over functionality. Apart from that, the use cases were generally the same.

For Logpoints, which only two of our participants ever used before, they could not remember the circumstances of their usage anymore.

Five participants stated that they use Conditional Breakpoints within loops to get to a specific iteration. One person additionally stated that they use it as some kind of sanity check to verify their mental model of the program. One participant mentioned that they only used it once in their school Computer Science class.

Regarding Hit Count Breakpoints, one participant could not remember the circumstances of their usage. The other two participants recalled that they used them in some special loops that contained bugs in only one or few iterations such that a hit count could be useful. One participant also mentioned that they used it within recursive functions.

Tool Opinions

> *"It feels wrong."*
>
> *—P12 about debugging with Print Statements*

Regarding Print Statements, some participants commented that it is not always possible to edit the source file to add them. Additionally, P11 noted that they often work with microcontrollers where no Print Statements are available. Multiple participants said that they think Print Statements are the first and most obvious thing to do.

Concerning Breakpoints, many participants praised the functionality of stepping. However, the usability of the Variable Viewer was disputed. While P7, P12, P14, and P15 mentioned that they liked the overview of all variables in the current context, P2, P3, P10, and P11 said that it can be "overwhelming" and "cluttery". P9 mentioned that they find it annoying that there is no "back" button when they step too far. Additionally, P2 experienced annoyances while debugging loops. They said that often, only one specific iteration of a loop is interesting so they have to use the Step Over feature many times to reach the iteration, sometimes overshooting and needing to restart. P18 opined that they do not need Breakpoints because Print Statements suffice.

> *"I am going to use them instead of Print Statements in the future!"*
>
> *—P7 about Logpoints*

P7 felt great appreciation towards Logpoints. Almost all participants noted that they found Logpoints easy to use and "like Print Statements, but better". They do not "trash the code" (P14). P11 also liked that the logging message does not mix with the terminal output because VSCode prints it to a separate output channel. Many participants said that they were shocked to learn that Logpoints are a part of VSCode because they never noticed it. P16 even

suspected it was implemented by us via a VSCode extension just for the user study.

> *"I prefer to use simpler tools. One should always think more about the bug than about the tool itself."*
>
> *—P4 about Conditional Breakpoints*

> *"Super useful."*
>
> *—P13 about Conditional Breakpoints*

Conditional Breakpoints were disputed among our participants. Many participants praised the ability to specify a condition, but some also said that they consider it a complicated task to devise a useful condition that covers all desired cases. P4 opined that it often costs too much time to think about the condition.

Many of our participants were critical of Hit Count Breakpoints. P5 said that they can not think of a use case for this tool. Almost all participants mentioned that it is complicated to devise a fitting hit count. P4 said that the mental model is simple, but the usage is more complicated. Furthermore, they opined that Hit Count Breakpoints are often not applicable to most debugging situations.

4.5 Discussion

In this section, we will interpret and analyze the results that we obtained from the user study based on our posed research questions. To that end, we will also derive interpretations based on the qualitative results of our participants in the questionnaire.

Our results regarding tool difficulty show that a majority of our participants consider all of the presented tools at least *Easy* and no participant chose *Difficult* or *Very difficult* as options (see Figure 4.6). The most difficult tool is the Hit

Count Breakpoint, with four participants choosing the option *Neutral*. Together with the Wilcoxon tests (see Table 4.2), we can derive that most participants considered Print Statements easier than all of the other tools.

The results in helpfulness illustrate that there are significant differences between Print Statements and Breakpoints, as well as between Hit Count Breakpoints and all the other tools (see Table 4.2). This, together with our qualitative results, indicates that most people rate Breakpoints significantly higher than Print Statements.

Our participants were most confident with Print Statements (see Figure 4.8). The Wilcoxon results (see Table 4.2) indicate a significance in the confidence assessment, similar to difficulty.

With our qualitative results, we can derive that most participants did not see sufficient use cases for Hit Count Breakpoints. For Conditional Breakpoints, many participants stated that, although they praise the functionality, they are deterred from using them because they find it difficult to find a useful condition.

Print Statements were praised by our participants for their simplicity although they rated them significantly worse in helpfulness compared to the other tools. The next tool our participants use after using Print Statements seems to be the Breakpoint, as it is rated significantly better than the Print Statement. Our participants also stated that they use Breakpoints whenever Print Statements do not suffice anymore.

4.5.1 Conclusions

Regarding our research questions, we can answer that all participants were familiar with Print Statements and Breakpoints. All the other tools were less known with Conditional Breakpoints being known by eleven participants and Hit Count Breakpoints and Logpoints being known by only five participants each.

Furthermore, we can deduct via our qualitative feedback from our participants that Print Statements are considered the easiest, most versatile tool. It is used for program flow analysis and variable inspection. Breakpoints come into use whenever Print Statements do not suffice, as in complex program structures. They are used because they provide the functionality of Step Over which replaces the need to use multiple Print Statements. However, many of our participants like this functionality but are often not willing to replace the usage of Print Statements with Breakpoints, because they are easy to use. Logpoints were not known by the majority of our participants. Conditional Breakpoints and Hit Count Breakpoints are almost exclusively used in loops by our participants. Hit Count Breakpoints even less, because they are not as powerful as Conditional Breakpoints and are only useful in specific circumstances.

Additionally, all our participants rated all tools except Print Statements and Hit Count Breakpoints almost exclusively with at least *Neutral*, with the majority rating them at least *Helpful*. Print Statements and Hit Count Breakpoints were rated significantly worse in their helpfulness.

To answer our last research question, we need to take a look at our qualitative results. Some participants felt that Print Statements alone did not suffice, but the advanced debugging methods were unknown to them. One participant stated that they will be using Logpoints more often from now on, indicating frustration with the usage of Print Statements.

Chapter 5

Summary & Future Work

In this thesis, we investigated how users debug and when exactly they prefer one debugging tool over another.

5.1 Summary & Contributions

We created the VSCode extension *DebugLog* that logs a developer's debugging behavior. Preliminary tests that we conducted apart from this thesis showed that the *DebugLog* extension has successfully logged multiple debugging sessions.

Furthermore, we conducted a user study to find out what developers think about different debugging tools, how often they use them, and in which situations they use them. The results showed that while the participants appreciate the functionalities of the tools, they still use Print Statements because it is the most versatile option. More advanced tools have a more specific use case or are simply not known. Breakpoints on the other hand are the second most used tool given its versatility because of its Stepping functionality being more applicable to a broader range of situations.

5.2 Future Work

In the future, *DebugLog* can provide a useful basis for research in the field of empirical IDE debugging. The features that it provides give useful insights into the developer's debugging technique. The logs provide an overview of how long a developer takes for each step and what tools they use for debugging different problems.

The results of our user study indicate that most developers do not know the advanced tools presented in this thesis. We conclude that these tools should be more advertised in the IDEs. However, Print Statements are still the lowest common denominator and are available in all modern programming languages. This may be the cause of its widespread usage in debugging.

Appendix A

Informed Consent Form

This appendix includes the Informed Consent Form that our participants had to read and sign before participating in our user study.

Informed Consent Form

User Study: Debugging in IDEs

Principal investigator: Lukas Liesenberg
RWTH Aachen University
Email: lukas.liesenberg@rwth-aachen.de

Purpose: The goal of this study is to obtain insights about different methods to find errors in python code.

Procedure: The study is conducted in person. Participating in this study involves interacting with an integrated development environment (IDE) to find and fix errors in Python code. We will collect general demographic information and ask general questions about the experience. A screen capture of you interacting with the IDE is recorded. The recording will also include your voice. You will be notified when the recording starts.
All information will be confidential. (See 'Confidentiality' below for details.)

Risks: The study should take approximately one to one and a half hours. Beside possible exhaustion or tiredness, there are no known risks. You can take breaks or abort at any time.

Confidentiality: All information gathered during the study will be kept confidential. You will be identified only through identification numbers and background information you divulge in publications, theses, or reports (pseudonymization). Please be aware that your voice can be heard in the screen recording, but you are not filmed. The recording will not be publicised.

Costs and Compensations: Participation in this study is voluntary. You are free to withdraw or discontinue the participation. Participation in this study will involve no cost to you and there will be no financial compensation. During the study sweets and drinks are available.

☐ I have read and understood the information on this form.
☐ I have had the information on this form explained to me.
☐ I accept the recording of my screen interactions and voice during the study.

Participant's Name	Participant's Signature	Date
	Principal Investigator	Date

Figure A.1: The Informed Consent Form that our participants needed to sign

Appendix B

User Study Questionnaires

This appendix includes the questionnaires that our participants received. These questionnaires consisted of one demographics questionnaire and five questionnaires related to debugging tools, one for each of the five tools that we chose for our user study.

The questionnaires that dealt with the debugging tools were conducted in an interview-like manner in which the investigator posed questions to the participant. The individual questionnaires for each debugging tool did not differ from one another and the same questions were asked for each tool.

During the the user study, the participants were also provided with an overview of all the shown tools.

ID: Phase 0: Demographics 1/3

Age: ___________________________

Gender: ___________________________

Profession / Course of Study: ___________________________

For how long have you been in this profession / course of study (in years):
____

Have you used integrated development environments (IDEs) in your profession or course of study?
☐ Yes
☐ No

For how long have you been using IDEs (in years): _______________________

Name IDEs you have experience with:

Name features of IDEs you have experience with:

Do you have experience in Python development?
☐ Yes
☐ No

For how long have you been using Python (in years): _____________________

Figure B.1: User Study Questionnaire, demographics part – Page 1

ID: Phase 0: Demographics 2/3

Have you used debugging tools integrated into IDEs?
☐ Yes
☐ No

For how long have you been using debugging tools integrated into IDEs (in years): ______________________

Name some debugging tools integrated into IDEs you have experience with:

Name some debugging tools for Python you have experience with:

Describe your typical debugging process inside of an IDE to us:

Figure B.2: User Study Questionnaire, demographics part – Page 2

ID: Phase 0: Demographics 3/3

How would you rate your typical debugging process?

Very Easy	Easy	Neutral	Difficult	Very Difficult
☐	☐	☐	☐	☐

How satisfied are you with your typical debugging process?

Very Satisfied	Satisfied	Neutral	Unsatisfied	Very Unsatisfied
☐	☐	☐	☐	☐

How confident are you within your typical debugging process?

Very Confident	Confident	Neutral	Unconfident	Very Unconfident
☐	☐	☐	☐	☐

Figure B.3: User Study Questionnaire, demographics part – Page 3

Phase 2

Print Statement
Prints onto standard output channel. Example: print("hello")

Breakpoint
Stop the program at a specified line and enable yourself to step through the program and view variable values.

Logpoint
Does not stop the program but prints output onto a specific output channel when its line is executed.

Conditional Breakpoint
Stops the program like a breakpoint but only if the specified condition is true.

Hitcount Breakpoint
Stops the program like a breakpoint but only after its line has been executed a specified amount of times.

Figure B.4: User Study Tool Overview

ID: Phase 2 - Tool:

Tool known?
☐ Yes
☐ No

Tool used before?
☐ Yes
☐ No

How often during debugging:
☐ Never
☐ Once
☐ Every few sessions
☐ Almost all sessions
☐ Always

Situations used in:

How easy to use?

Very Easy	Easy	Neutral	Difficult	Very Difficult
☐	☐	☐	☐	☐

How helpful?

Very Helpful	Helpful	Neutral	Unhelpful	Very Unhelpful
☐	☐	☐	☐	☐

How confident in using?

Very Confident	Confident	Neutral	Unconfident	Very Unconfident
☐	☐	☐	☐	☐

Additional information:

Figure B.5: User Study Questionnaire, tool part (one for each tool)

Bibliography

Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. On the dichotomy of debugging behavior among programmers. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 572–583, 2018. doi: 10.1145/3180155.3180175.

Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola. Code bubbles: a working set-based interface for code understanding and maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, page 2503–2512, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605589299. doi: 10.1145/1753326.1753706. URL https://doi.org/10.1145/1753326.1753706.

Ben Buhse, Thomas Wei, Zhiqiang Zang, Aleksandar Milicevic, and Milos Gligoric. Vedebug: Regression debugging tool for java. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 15–18, 2019. doi: 10.1109/ICSE-Companion.2019.00027.

Davor Čubranić and Gail C. Murphy. Hipikat: recommending pertinent software development artifacts. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, page 408–418, USA, 2003. IEEE Computer Society. ISBN 076951877X.

R. DeLine, M. Czerwinski, B. Meyers, G. Venolia, S. Drucker, and G. Robertson. Code thumbnails: Using spatial memory to navigate source code. In *Visual Lan-*

guages and Human-Centric Computing (VL/HCC'06), pages 11–18, 2006. doi: 10.1109/VLHCC.2006.14.

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

Eduardo Andreetta Fontana and Fabio Petrillo. Mapping breakpoint types: an exploratory study. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pages 1014–1023, 2021. doi: 10.1109/QRS54544.2021.00110.

John D. Gould and Paul Drongowski. An exploratory study of computer program debugging. *Human Factors*, 16(3):258–277, 1974. doi: 10.1177/001872087401600308. URL `https://doi.org/10.1177/001872087401600308`.

W. Harrison and C. Cook. Insights on improving the maintenance process through software measurement. In *Proceedings. Conference on Software Maintenance 1990*, pages 37–45, 1990. doi: 10.1109/ICSM.1990.131320.

Thorsten Karrer, Jan-Peter Krämer, Jonathan Diehl, Björn Hartmann, and Jan Borchers. Stacksplorer: call graph navigation helps increasing code maintenance efficiency. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, page 217–224, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450307161. doi: 10.1145/2047196.2047225. URL `https://doi.org/10.1145/2047196.2047225`.

Brian W. Kernighan. *UNIX for Beginners*, volume 1. Bell Laboratories MURRAY Hill, NJ, 1978.

Brian W. Kernighan and Phillip J. Plauger. *The elements of programming style / Brian W. Kernighan ; P. J. Plauger*. McGraw-Hill, New York [u.a, 2. ed. edition, 1978. ISBN 0070342075.

Mik Kersten and Gail C. Murphy. Mylar: a degree-of-interest model for ides. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, AOSD '05, page 159–168, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930426.

doi: 10.1145/1052898.1052912. URL `https://doi.org/10.1145/1052898.1052912`.

Amy J. Ko and Brad A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, page 301–310, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605580791. doi: 10.1145/1368088.1368130. URL `https://doi.org/10.1145/1368088.1368130`.

Amy J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on software engineering*, 32(12):971–987, 2006.

Jan-Peter Krämer, Joachim Kurz, Thorsten Karrer, and Jan Borchers. Blaze: supporting two-phased call graph navigation in source code. In *CHI '12 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '12, page 2195–2200, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450310161. doi: 10.1145/2212776.2223775. URL `https://doi.org/10.1145/2212776.2223775`.

Thomas D. LaToza and Brad A. Myers. Visualizing call graphs. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 117–124, 2011. doi: 10.1109/VLHCC.2011.6070388.

Henry Lieberman. The debugging scandal and what to do about it. *Communications of the ACM*, 40(4):26–30, 1997.

Paul W Oman, R Cook, Murthi Nanja, et al. Effects of programming experience in debugging semantic errors. *Journal of Systems and Software*, 9(3):197–207, 1989.

Martin P Robillard, Wesley Coelho, and Gail C Murphy. How effective developers investigate source code: An exploratory study. *IEEE Transactions on software engineering*, 30(12):889–903, 2004.

Benjamin Siegmund, Michael Perscheid, Marcel Taeumel, and Robert Hirschfeld. Studying the advancement in

debugging practice of professional software developers. In *2014 IEEE International Symposium on Software Reliability Engineering Workshops*, pages 269–274, 2014. doi: 10.1109/ISSREW.2014.36.

J. Singer, R. Elves, and M.-A. Storey. Navtracks: supporting navigation in software maintenance. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 325–334, 2005. doi: 10.1109/ICSM.2005.66.

Richard Stallman, Roland Pesch, Stan Shebs, et al. Debugging with gdb. *Free Software Foundation*, 675, 1988.

Andreas Zeller. *Why programs fail: a guide to systematic debugging*. Elsevier, 2009.

Qin Zhao, Rodric Rabbah, Saman Amarasinghe, Larry Rudolph, and Weng-Fai Wong. How to do a million watchpoints: Efficient debugging using dynamic instrumentation. In *Compiler Construction: 17th International Conference, CC 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 17*, pages 147–162. Springer, 2008.

Index

breakpoint 1, 2, 9
breakpoint taxonomy 7, 20

code navigation 5
conditional breakpoint 8, 20, 27–38

extension 11–18

future work 40

hit count breakpoint 20, 27–38

line breakpoint 8, 20
logpoint 20, 27–38

print statement 2, 20, 27–40

stack trace 2
summary 39

tracepoint 9

user study 19–38

variable viewer 2
Visual Studio Code 2, 6, 9, 11–13, 17, 20, 39
VSCode . . . . . *see* Visual Studio Code

Typeset March 27, 2024