What Programmers Know About Debugging And How They Use Their IDE Debuggers

Master's Thesis

Niels Spruit

What Programmers Know About Debugging And How They Use Their IDE Debuggers

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Niels Spruit
born in Nieuwegein, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

What Programmers Know About Debugging And How They Use Their IDE Debuggers

Author: Niels Spruit
Student id: 4137639
Email: N.Spruit@student.tudelft.nl

Abstract

As new bugs are discovered continuously, software developers often face the task of locating and fixing the defect causing the failure, called debugging. Based on the absence of behavioral studies on this subject in literature, this study aims to get more insights into how developers think of debugging and how they debug in their IDE. To this end, after searching for common issues with debugging on StackOverflow, we conducted an online survey on developers' perception on debugging. In addition, we developed a plugin to instrument Eclipse and IntelliJ in order to look for common debugging behavior. Amongst others, we found that while the vast majority of survey respondents claims to be using the IDE debugger, most plugin users actually do not use it. Furthermore, we found that the amount of testing performed or programming experience has limited to no impact on the time spent debugging. In general, the results give a strong indication that we need to review some commonly accepted beliefs on debugging.

Thesis Committee:

Chair:	Dr. A.E. Zaidman, Faculty EEMCS, TU Delft
University supervisor:	M.M. Beller, Faculty EEMCS, TU Delft
Committee Member:	Dr. C. Hauff, Faculty EEMCS, TU Delft
Committee Member:	Dr. M.B. van Riemsdijk, Faculty EEMCS, TU Delft

Preface

This thesis report is the product of my graduation project that was performed as part of my Master Computer Science at Delft University of Technology. However, this work could not have been completed without the help of several people. First of all, I would like to thank the anonymous survey respondents and WatchDog 2.0 users for generating the data used for deriving the results of this study. Next, I would like to thank Andy Zaidman, Georgios Gousios, Moritz Beller, Annibale Panichella and Igor Levaja of the TestRoots team for developing WatchDog, which served as a great basis for this project. In particular, I want to thank Moritz Beller for his critical, but constructive feedback throughout the process and for his time spend on reviewing my code. Next, I would like to thank Andy Zaidman for this guidance and feedback throughout this project. In addition, I would like to thank Verstoep Bouwadvies BV for providing me with an inspiring working environment during the entire graduation project. Finally, I would like to thank my family and friends for their unconditional support throughout my thesis as well as my study in general.

Niels Spruit
Delft, the Netherlands
July 17, 2016

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Since the introduction of computer science, bugs seem to have become part of our daily lives. New bugs are discovered on a daily basis and some bugs even make it to the (inter)national news. Examples of such bugs include the recent 'heartbleed bug' that posed a threat on the security of the internet, enabling hackers to retrieve personal information and passwords from users [50]. Unfortunately, even more severe bugs occur every once in a while as well. For instance, after ten years of development and an investment of seven billion dollars, the Ariane 5 rocket crashed within a minute after its launch due to a software bug [14]. Moreover, software bugs have even been responsible for human casualties. An example of such a bug is the bug in the Patriot missile defence system that resulted in 28 American casualties in 1991 [43].

To prevent bugs from causing even more damage, software developers are faced with the task of fixing the defect causing the failure, called debugging. However, Henry Lieberman stated that debugging "is the dirty little secret of computer science" [22]. This statement can be considered confirmed when you consider the fact that testing and debugging activities consume $30-90\%$ of the labour in a project [3] and cost the U.S. about $59.5 billion a year [48].

Based on the absence of behavioral studies on this subject in literature, the aim of this study is to get more insights into how developers think of debugging and how they debug in their Integrated Development Environment (IDE). To steer our research, we introduce the following three research questions:

RQ1 What is developers' self-assessed knowledge and conception of debugging?

RQ2 When and how long do developers debug?

RQ3 What is typical behavior during debugging?

To make (the analysis for) the last two research questions more concrete, we also define the following set of sub research questions:

RQ2.1 How much of their active IDE time do developers spend on debugging (compared to other activities)?

RQ2.2 What is the (average) frequency and length of the debugging sessions performed by developers?

RQ2.3 At what times do developers launch the IDE debugger?

RQ2.4 Do long files require more debugging?

RQ2.5 Is a small set of classes responsible for most debugging effort?

RQ2.6 Do developers who test a lot have to debug less?

RQ2.7 Do experienced developers have to debug less?

RQ3.1 Which sequences of events are commonly found within and between debugging sessions?

RQ3.2 How do breakpoints evolve over time?

RQ3.3 How often does it occur that a developer steps over the point of interest and has to start all over again?

To find the answers to these research questions, we first sketch the context of this study and review the available literature on this subject in Chapter 2. To answer **RQ1** we developed and conducted an online survey on the perception of debugging, of which the design and results are described in Chapter 4. However, before designing this survey, we investigate whether there are common issues with debugging amongst developers in Chapter 3 by focussing on the data of the popular Q&A site StackOverflow[1]. Using this data, we might identify commonly asked questions about debugging which would serve as an indicator of the common debugging issues, which could form the basis of our survey.

Next, we extended the "research vehicle that tracks the testing habits of developers" [6], WatchDog, to also track developers' debugging behavior in order to find answers to **RQ2** and **RQ3** as well as their sub research questions listed above. The extension of WatchDog to the new version, WatchDog 2.0, itself is described in more detail in Chapter 5 and in Chapter 6 we describe how we collect and analyse the data obtained with WatchDog 2.0 as well as the results of these analyses.

In Chapter 7 we discuss the results obtained using the survey and the WatchDog 2.0 data on a more abstract level and we discuss possible threats to the validity of our study. Finally, we define this study's conclusions and implications as well as directions for future research in Chapter 8.

[1] http://stackoverflow.com/

Chapter 2

Background and Related Work

This chapter begins with an introduction to the general debugging process, followed by a description of important debugging techniques and debugging in the Integrated Development Environment (IDE). Next, we describe existing research studying the debugging practice of developers. Finally, we describe tools that are related in concept to WatchDog 2.0 and provide a short introduction to the natural language processing that is required for analysing the StackOverflow data.

2.1 Debugging process

To reduce the time spent on finding and fixing a defect that causes a program failure, a systematic approach to debugging can be useful. An example of such an approach is *TRAFFIC* by Zeller [55]. It comprises seven steps that cover any action required in the debugging process, from the moment a problem is discovered until the moment the defect causing the failure is corrected. The seven steps belonging to this approach are:

1. **T**rack the problem in the database
2. **R**eproduce the failure
3. **A**utomate and simplify the test case
4. **F**ind possible infection origins
5. **F**ocus on the most likely origins
6. **I**solate the infection chain
7. **C**orrect the defect

Steps 4 to 6 of the TRAFFIC approach comprise the often called Find-Focus-Isolate loop, which is "by far the most time consuming" as these steps often need to be applied iteratively in order to find the (root) cause of the failure. Therefore, much research has gone into techniques to (partially) automate the steps in this loop in order to reduce the effort required for debugging, which we will discuss next.

2.2 Debugging techniques

One of the techniques that can be used in the Find-Focus-Isolate loop is called *delta debugging*, which can be used to systematically narrow down possible infection origins by comparing two program runs, a failing and a passing one [54]. Several variants of and improvements to delta debugging are described in literature as well, e.g. in [21], [47], [53] and [55].

Another technique operates by following the dependencies in the dependence graph of the program so that a developer can focus on only a subset of the statements, called a *slice*, that could have caused an infected program state. This technique called *slicing* can be used in two basic ways: for creating a *forward slice* and for creating a *backward slice* [55]. Several improvements to slicing have also been developed, see e.g. [58] and [59].

Another approach that is described by Zeller is to focus on *anomalies*, meaning that you compare the behavior of an execution to the normal program behavior. Although abnormal behavior does not imply incorrect behavior, it is often "a good indicator of defects" and it therefore "wise [...] to focus on anomalies for further observation or assertion" [55]. Examples of techniques related to this approach can be found in e.g. [2] and [30].

The fourth type of techniques is based on *mining dynamic call graphs*. A dynamic call graph is a "concise representation of a programme execution and reflects the method-invocation structure" [11]. Techniques based on mining these graphs can also be found in literature, see e.g. [11] and [12].

Another type of techniques is called *statistical debugging*, which uses statistical methods to extract a model of the program based on the value of program predicates as branches, loops and return values in different program executions. Then, these models are used to identify faults by detecting misbehaviors in the program [31]. Examples of statistical debugging techniques are described in e.g. [31] and [32].

A subtype of statistical debugging is *spectra-based fault localization*. By instrumenting programs, data about which statements are executed and which not can be collected. Then, a "summary of this data, often called program spectra, can be used to rank the parts of the program according to how likely it is they contain a bug" [27]. This ranking of program parts is based on a so-called 'ranking metric': a numeric function that is applied to collected spectra. An evaluation and comparison of many different ranking metrics can be found in [23].

The list of types of techniques mentioned above is by far not exhaustive. Other types of techniques described in literature include *angelic debugging* [9], *data structure repair* [24], *relative debugging* [1], *automatic breakpoint generation* [57] and *automatic program fixing using contracts* [51]. Finally, several combinations of techniques exists as well, see e.g. [20], [33], [34], [42] and [52].

Since these automated debugging techniques are not often found in popular IDEs and our study focuses on debugging inside the IDE, we do not discuss them in more detail in this report. For a more detailed explanation and evaluation of these techniques, see e.g. the talk on automated debugging techniques by Orso [29].

2.3 Debugging in the IDE

Besides using the techniques described above, developers can also rely on *symbolic debuggers* like GDB [46] throughout the debugging process. Such debuggers allow developers to specify points in the program where the execution should be suspended, called *breakpoints*. A typical symbolic debugger supports different types of breakpoints and options to further refine the moment when the program should be suspended, e.g. by specifying a condition that indicates if the execution should actually be suspended at a breakpoint hit.

When the program is suspended, developers can use symbolic debuggers to, e.g., inspect variables and the call stack, step through the code and evaluate expressions [46]. Originating from commandline symbolic debuggers like GDB, several graphical symbolic debuggers have been developed, e.g. DDD [56]. Most of the symbolic debugging features have nowadays been integrated in Integrated Development Environments (IDEs) like Eclipse and IntelliJ. This study focusses on such IDE debuggers and how developers use them.

2.4 Debugging practice

To see to what extent the methods and techniques described above have been adopted in the real world, Siegmund et al. studied the debugging practices of professional software developers [44]. They followed eight software developers across four different companies during "the course of their day" by observing their methods by letting them think aloud and answer several questions. Their results indicate that none of the developers had any debugging-specific education or training. Furthermore, "all developers use a simplified scientific method", which consists of formulating and verifying hypotheses as described by Zeller [55]. Moreover, they found that "all participants are proficient in using symbolic debuggers" and also prefer them to using print statements. Finally, they found that none of the developers was aware of back-in-time debuggers and the techniques described above as well as more advanced symbolic debugging features, such as conditional breakpoints.

Subsequently, Siegmund et al. created an online survey on "debugging tools, workload, approach and education" [35]. Based on the answers of 303 respondents, they found that "debugging education is still uncommon, but more [...] courses started including it recently" as there is also "a need for most developers to learn more about debugging in general". Furthermore, most participants only use "older debugging tools such as printf, assertions and symbolic debuggers" while there is "almost no correlation between the programming language used and the debugging tools used". Finally, concurrency issues and external libraries often seem to be the root causes of the hardest bugs. While their study focusses mainly on debugging in general, our study is aimed specifically at debugging within the IDE.

Piorkowski et al. studied qualitatively how programmers forage for information [38, 39] and provided a recommendation system for navigation during debugging sessions [37]. They found that developers foraged for information remarkably different when they were asked to fix a bug versus when they had to learn enough to fix a bug, "[d]espite the subtlety of difference". They also found that developers spent half of their debugging time foraging for information. This relates to our study as it shows what parts of the IDE are often used

for gathering information during debugging, which might have an impact on the set of debugging features being used.

2.5 Related tools

In [36], Petrillo et al. develop the *Swarm Debug Infrastructure* (SDI), which "provides [Eclipse] tools for collecting, sharing, and retrieving debugging data". Developers can then utilize the collective swarm intelligence of previous debug sessions to "navigate sequences of invocation methods" and "find suitable breakpoints". Petrillo et al. evaluated SDI in a controlled experiment, in which 10 developers were asked to debug into three faults of the open-source software JabRef. They found that developers toggled only one or two breakpoints per fault, that there were no correlations between numbers of breakpoints and developers' expertise and task complexity. Instead developers followed diverging debugging patterns.

Our approach for **RQ2** is technically similar to SDI in that we instrument Eclipse. In order to broaden the generalizability of our findings, we also support a second IDE, IntelliJ, and performed a longitudinal field study of how dozens of developers debug in the wild. To this end, we built on the existing WatchDog infrastructure by adding debug monitoring capabilities for both Eclipse and IntelliJ in WatchDog 2.0. Beller et al. originally introduced WatchDog to verify common expectations and beliefs about testing by instrumenting developers' IDEs and thus objectively monitoring their testing behavior [4]. More on the extension of WatchDog to WatchDog 2.0 is described later in this report.

2.6 Topic modelling

To organize and summarize large volumes of unstructured text that "would be impossible by human annotation" [7], automated techniques might come to the rescue. A technique that is often applied in such situations is *Latent Dirichlet Allocation* or simply *LDA* [8]. In this technique, *Latent* refers to the assumption that the "cluster of words exist and are responsible [...] for the frequencies observed" in the collection of unstructured text and *Dirichlet* points to the probabilistic distribution that is used to infer the distribution of the cluster of words in the text [41]. These clusters of words are often called *topics*. Therefore, the probabilistic process described above is commonly referred to as *topic modelling*, of which LDA is just one example.

This relates to our study as the collection of StackOverflow questions we want to analyse for common debugging topics is too large for manual annotation. Therefore, an automated method is required to perform this annotation for us. As the StackOverflow questions are unstructured, we decided to use LDA as it is "useful for analyzing large collections of unlabelled text" [26]. In particular, we used the tool MALLET [26], as it is open source and it fits the needs of this study. The detailed process used for this investigation and its results will be discussed next in the following chapter.

Chapter 3

Common Debugging Issues on StackOverflow

This chapter is structured as follows. First, the process of selecting and preprocessing the input data to be used for topic modelling the StackOverflow questions is addressed in Section 3.1. Next, the remaining sections describe the results of the topic modelling process for a particular selection of posts that serves as the input for the process.

3.1 Data selection and preprocessing

The StackOverflow data used for this research is taken from the data dump as provided on the Mining Software Repositories Conference website[1]. This data dump contains all StackOverflow data up to September 26, 2014. To be able to query this data, we first imported it into an SQL database using a script written by Georgios Gousios and Megan Squire[2].

By querying the resulting database, a few basic statistics can be extracted. First of all, the entire data dump contains 21,736,594 posts. In StackOverflow's data model, posts can be either questions or answers. As only questions are required for this research, we filtered out all answers. In total the data dump contains 7,990,787 questions, which is about 37% of all posts. However, not all of these questions are relevant for the research described in this chapter. Therefore, the following SQL query was used:

```
SELECT *
FROM posts
WHERE PostTypeId = 1
AND (Title LIKE '%debug%'
OR Body LIKE '%debug%');
```

The query above retrieves all questions of which the title and/or the question itself contains the phrase 'debug'. In this way, questions containing words like 'debugger' and 'de-

[1] http://2015.msrconf.org/challenge_data/

[2] https://gist.github.com/megansquire/877e028504c92e94192d

bugging' are selected as well. This query results in a total of 281,562 posts, which means that about 3.5% of the questions found on StackOverflow contain the phrase 'debug' and are therefore likely about debugging. Finally, we exported these results to a CSV file for further processing as described next.

As MALLET mainly works with text (.txt) files as input, the CSV file resulting from the SQL query needed to be converted. Furthermore, to be able to trace back topics to questions, a separate text file had to be created for each question. To accomplish this, we created a Python script to extract the title and the body of each question from the CSV file, as these are the only two fields that are useful for the analysis. In addition, this script first unescapes all escaped characters and strips the HTML tags that are found in the question's body before writing the title and body to a text file. At this point, all resulting text files could be imported into MALLET to use them as input for the topic modelling process.

3.2 Topic modelling with all posts

To get an idea of which common topics can be identified in the StackOverflow questions, we first executed MALLET's topic modelling function using all posts as input. To avoid having most of the questions in a quite limited number of topics, this was done with both 20 and 50 topics as output. The resulting topic model with 20 topics is shown in Table 3.1. This table not only contains the key words and weight belonging to a particular topic, but it is also extended with a column that aims to give a representative label that describes the topic based on its key words. Due to spacial concerns, a similar table for the 50-topic case can be found in Section A.1 of Appendix A.

As can be seen in Table 3.1, most topics resulting from the topic modelling process correspond to issues with a particular programming language or technology. Moreover, none of the resulting topics gives a clear indication of the existence of common issues with debugging, as the topics that are actually about debugging, e.g. 'GDB issues' and 'General debugging', are not specific enough to pinpoint to a specific problem with debugging. Increasing the number of resulting topics to 50 does not achieve this as well, as the results are quite similar to the ones with 20 topics.

3.3 Topic modelling with 'Java posts'

To see whether making the input documents more specific makes the resulting topics more concrete as well, we also created a topic model for a subset of the input documents, so for a subset of the questions. For this section, the goal was to create a topic model with the documents belonging to the two Java topics in Table 3.1, 'Java/C# development' and 'Java/Node.js errors', as input. However, selecting all documents belonging to a particular topic is not possible in MALLET, as each document can belong to multiple topics with a certain weight for each topic.

Therefore, an alternative solution is necessary to accomplish a similar goal. As MALLET produces a composition file that for each input document contains the weight of each resulting topic, we used these weights to find documents that belong most to a certain topic.

Topic #	Weight	Key words	Possible descriptive label
0	0,03044	request response data url json var post error http code string api server return function facebook token user status true	HTTP errors
1	0,02752	int return void float array double code function char null include node vector cout size data number case program const	C/C++ development
2	0,06238	public class string return void private null static object method set int import list exception code catch override var type	Java/C# development
3	0,01663	error nsstring ios app view alloc xcode nil fff iphone code return framework data init armv property method simulator corefoundation	iOS development
4	0,04548	end string table query database dim data sql select set null code error date row values function column integer insert	SQL errors
5	0,01508	public e/androidruntime import void override private null activity string int method return class final intent android view static error/androidruntime extends	Android development
6	0,37785	code debug i'm application problem file debugging i've run project app error work time running works it's don't debugger line	General debugging
7	0,05283	server service client error debug connection web message wcf port connect socket host request iis file remote http application configuration	WCF/IIS issues
8	0,01892	debug file error install checking directory version warning command make gcc build found ssh php configure failed files library running	Make compilation issues
9	0,02245	i/debug thread int gdb memory char code return bytes kernel program data null stack include address mov info/debug function size	GDB issues
10	0,02751	array user email echo true php function password error mail return false username login debug message form session line smtp	PHP user authentication
11	0,00877	gem call end actionpack rails app[web block ruby require activesupport vendor/bundle/ruby def rack error run activerecord true railties lib/ruby gems/actionpack	Ruby errors
12	0,01288	debug npm err com.opensymphony.xwork http error info spring bean exception null method hibernate defaultactioninvocation.invoke(defaultactioninvocation.java session main source filter true request	Java/Node.js errors
13	0,01697	line file import python def return print django true error class debug call module traceback false app url main c:\python	Python errors
14	0,0227	info error log debug jar file java project build source compile maven class failed android plugin run files eclipse true	Maven building issues
15	0,02744	var true text width event button height false void item private code form image sender color control function set window	Javascript event handling
16	0,02768	file string image byte null stream data filename upload files code read catch video int key path return size bytes	File I/O issues
17	0,01855	error include const function int void lnk symbol undefined reference char studio class unresolved file external referenced lib return files	C/C++ development
18	0,06756	function var page code javascript html jquery data return url error view form ajax true i'm chrome controller object button	Web development (front-end)
19	0,02318	dll exception assembly loaded error file version boolean object int win code net type studio application symbols string null intptr	.NET Windows errors

Table 3.1: Topic modelling results with all posts and 20 topics.

Using this information, all documents can be found in which one of the two Java topics is the topic with the largest weight of all topics. Doing this for the two Java topics using another Python script results in a subset of 26,348 questions.

Applying the topic modelling function to these documents as input and 20 topics as output results in the table shown in Table 3.2. For the same reasons as described in the previous section, a topic model was also created with 10 and 30 topics as output. The results for these numbers of topics can again be found in Appendix A in Section A.2 and A.3, respectively.

Looking at the 20 resulting topics for the 'Java posts' in Table 3.2, we can draw a similar conclusion to the one of the previous section. Also when using this particular subset of questions, most topics still correspond to problems with particular programming languages, environments and technologies. Furthermore, no specific issues with debugging can be extracted from the resulting topics as well. The same conclusion can also be drawn from the results of the other two settings for the number of topics. Therefore, making the input more specific by selecting only the 'Java posts' does not make the resulting topics more general and abstract. This suggests that making this input even more specific seems to be of little use.

3.4 Topic modelling with 'general posts'

Another topic in the topic model resulting from taking all posts as input that might be interesting for further inspection is 'General debugging'. To see whether using only the documents about this topic as input makes the results more specific, we performed a similar procedure to the one described above. This process resulted in a total of 71,726 StackOverflow questions.

For the same reasons as before, we created a topic model for these 'General posts' with 10, 20 and 30 topics. The results for 20 topics are shown in Table 3.3 and the results for 10 and 30 topics can again be found in Section A.4 and A.5 of Appendix A, respectively.

In contradiction to the results for the 'Java posts' described in the previous section, the results for the 'General posts' are actually more specific as can be seen in Table 3.3 and Section A.4 and A.5. For instance, the topics 'Breakpoint issues', 'Debugging Java applications in the IDE' and 'Watching variable values while debugging' in Section A.5 are quite more specific than 'General debugging' and might also pinpoint to more specific common issues with debugging. So, in this case making the input documents more specific makes the resulting topics more general and abstract as well. However, the resulting topics are still not specific enough to pinpoint concrete debugging issues.

Therefore, we applied the same process again for the three more specific topics mentioned above, but now with the 'General posts' serving as basis. In this way, the question was whether or not this would result in even more specific and maybe even concrete debugging issues. This process resulted in inputs consisting of 3,208, 2,535 and 2,340 questions for the topics 'Breakpoint issues', 'Debugging Java applications in the IDE' and 'Watching variable values while debugging', respectively.

Topic #	Weight	Key words	Possible descriptive label
0	0,05837	service server client exception thread error wcf connection message web task code socket request catch application var string iis void	WCF/IIS errors
1	0,01339	loadmodule php server file apache error default root load configuration options include directory notice files client version directive set enabled	PHP configuration errors
2	0,01193	e/androidruntime i/debug error/androidruntime android d/dalvikvm method freed paused activity w/system.err free exception i/system.out error main info/debug pid thread failed device	Android errors
3	0,02201	file line error return import true files django upload call python filename request string image copied response path traceback envelope	Python errors
4	0,0242	info error debug build project jar files compile file java version maven android failed plugin package source test found class	Maven compilation errors
5	0,01668	error include void int const lnk studio dll referenced visual loaded class unresolved return symbol function public c:\program external lib	Visual Studio errors
6	0,01394	view nsstring alloc nil error return code true ios init app cell release delegate data controller property false tableview files	iOS errors
7	0,01202	fff npm err error armv thread framework setenv node http ffff ios corefoundation app xcode verbose install uikit stack crash	iOS errors
8	0,05926	query database table array data select null sql list model return echo set date entity string values function row result	SQL development
9	0,01669	error checking file warning make build undefined gcc install directory found version reference copying include library gdb files target command	Gcc compilation
10	0,05741	user request error url server password response email post login return true page username http data session function mail message	Web authentication
11	0,04541	var function page true false return jquery data ajax javascript html width code image event button url height click text	Web development
12	0,00875	end gem call app[web actionpack block rails ruby c:/ruby vendor/bundle/ruby require lib/ruby/gems activesupport def lib/ruby rack true method render false	Ruby development
13	0,03768	int return char null void include function bytes size array code data program struct double break unsigned const node case	C/C++ development
14	0,02483	debug spring bean info error exception tomcat file session null type log configuration hibernate source key exec true servlet request	Tomcat/hibernate errors
15	0,02558	import log def print python debug class file logger return logging true line info main level error message script output	Python logging errors
16	0,03435	public void import private string null override int return final class activity static catch view extends android intent button method	Android development
17	0,02428	end dim string code set text function error row true form data integer private xml false table sender excel select	Excel development
18	0,14443	public class string return object method void private null set static code list type var int exception property test i'm	Java/C# development
19	0,41547	i'm code problem file i've debug application error run work app works time it's project don't fine set debugging working	General debugging

Table 3.2: Topic modelling results with Java posts and 20 topics.

Topic #	Weight	Key words	Possible descriptive label
0	0,08981	file files folder path directory error project copy application xml find open source created problem access location image load code	File I/O errors
1	0,05594	command script python debug run line file running shell program console print perl output i'm commands emacs module import set	Debugging scripts
2	0,06895	eclipse debug java application run project debugging plugin netbeans ide tomcat server debugger running source intellij remote class jar code	Debugging Java applications with a Tomcat server
3	0,05786	string code i'm line output problem text print result array number xml characters values strings input data wrong loop character	Issues with Strings/Arrays
4	0,06344	memory object variable variables debugger objects values debugging watch i'm data array window list view size type class debug access	Watching variable values while debugging
5	0,10319	time process thread application running problem run debug threads app seconds task program debugging start takes slow stop long issue	Timing issues with debugging
6	0,12663	code exception debugger line breakpoint debugging debug step break set stack error breakpoints exceptions point program source i'm call trace	Breakpoint issues
7	0,06225	form event button view code user control click custom method i'm problem text set window application list works called add	User Interface development issues
8	0,33567	i'm code debugging i've question find it's time don't make i'd good work lot things tool found tools answer simple	General debugging/Questions about tools
9	0,04118	database data sql query server connection table debug mysql stored queries entity i'm model sqlite string procedure select tables problem	SQL problems
10	0,0865	javascript page debug chrome browser i'm html firefox script debugging problem jquery working web code work css works issue tools	Debugging web applications in the browser
11	0,11785	server web application asp.net app service iis page error debug site request mvc i'm client works local problem url fine	Asp.NET/IIS errors
12	0,11008	project build debug release file library files solution configuration projects mode dll version set compile folder reference assembly add source	Debug/release build configuration issues
13	0,08082	error test tests run rails unit debug message failed errors i'm line ruby code problem report running fails i've issue	Ruby unit testing issues
14	0,06713	android app device phone debug emulator application eclipse usb debugging adb google devices run key windows apk sdk problem i'm	Eclipse Android debugging issues
15	0,06711	log php output debug logging console file messages logs xdebug i'm message trace level application error info net print logger	PHP logging
16	0,06321	app xcode ios debug iphone flash device application simulator run error build i'm fine mac version running flex ipad problem	Xcode iOS debugging issues
17	0,1572	visual studio debug application windows debugging project run dll net debugger start error running process problem program window bit machine	Visual Studio debugging problems
18	0,06217	gdb program debug linux windows debugging code debugger kernel i'm information run gcc symbols library dump crash bit file compile	GCC/GDB debugging issues
19	0,07114	function class code method public return call debug void int static called foo functions main object null define string methods	General development

Table 3.3: Topic modelling results with general posts and 20 topics.

The topic models resulting from taking these subsets as input and 20 topics as output can be found in Section A.6-A.8 of Appendix A. As can be seen in these tables, no descriptive label is provided as we could not attach any meaningful label to these topics based on the key words in the resulting topic model. Furthermore, no concrete issues with debugging can still be identified based on these key words. Therefore, continuing in this way would no longer make sense.

Chapter 4

Developers' Perception on Debugging

To get to know the developers' perception on debugging, this chapter first describes the design of the online survey we conducted and the methods used for analysing its responses. Next, we present and discuss the results of these analyses and their interpretations.

4.1 Research design and methodology

This section first describes how we set up the online survey and how we tried to attract as many developers as possible. Next, we describe the methods used for analysing the responses to this survey.

4.1.1 Survey design

To investigate developers' self-assessed knowledge and conception on debugging for **RQ1**, we set up an online survey[1], of which a printed version can be found in Appendix B. The survey consisted of four different question sections. The first collected general information about the respondents, such as their programming experience. The second part asked participants if and how they use the IDE-provided debugging infrastructure in general, as well as its particular features. We introduced a branching condition in the survey: Developers who do not use the IDE-provided debugging infrastructure were asked for the reason why, while others got questions on specific debugging features. These questions asked how well the respondent knows and uses several types of breakpoints. In addition, we asked questions about other debugging features ranging from stepping through code to more advanced features like editing code at runtime (hot swapping).

The third part, presented to all respondents again, assessed the position of (unit) testing in the debugging process. The three questions within this section asked if the developer uses tests for reproducing the bug, checking progress during the debugging process and/or to verify possible bug fixes. The last part consisted of an open, non-mandatory question asking the participants' opinion on the following statement: "the best invention in debugging still was printf debugging". Survey research has shown that asking a controversial, concrete

[1] `http://goo.gl/forms/WkLCAHmmFBMRupJE2`

statement evokes participants' emotions and leads to more insightful answers [10]. As we found no common debugging issues on StackOverflow, no questions were added based on that analysis. Before releasing the survey to the public, we made several internal iterations, sharpened the IDE focus, and ran it across six programmers not involved in its design.

To attract as many survey respondents as possible, we spread the link to the survey through social media, especially via Twitter. In addition, we contacted several people by email to ask them to retweet the link to reach an even larger audience. Finally, to create an incentive for developers to fill out our survey, we advertised that we will raffle of three 15 Euro Amazon vouchers among the survey respondents.

4.1.2 Analysis of survey responses

To analyse the obtained survey responses, we exported them and imported them into R for further analysis. First, we analysed the responses to each question in isolation, followed by an analysis of the relations between pairs of questions. For several dependency analyses, we used the *Spearman rank-order correlation test*, which we preferred over other alternatives as it is non-parametric, it works with both continuous and discrete (ordinal) variables and it does not only look at linear relationships [19].

For interpreting the results of these dependency analyses we performed using the Spearman rank-order correlation test, we use Hopkins' guidelines [16]. These guidelines tell that a value (ρ) resulting from the Spearman rank-order correlation test indicates no correlation for $0 \leq |\rho| < 0.3$, a weak correlation for $0.3 \leq |\rho| < 0.5$, a moderate correlation for $0.5 \leq |\rho| < 0.7$ and a strong correlation for $0.7 \leq |\rho| \leq 1$.

For the final analysis of the survey responses, we used the responses to the open question to perform an *open card sort* [45] in which the first two persons agreed on an initial, mutually shared set of tags, which we then used to tag individual responses. Finally, a third person sampled 20% of the responses and tagged them again independently, without seeing the tags assigned by the first two persons. Then, we converged our tag sets to arrive at a homogeneous classification.

4.2 Survey results and their interpretations

This section presents the results and their interpretations of the different analysis methods described above. First, we report on the results of the analysis of questions in isolation, followed by the dependency analysis of pairs of questions. Finally, we will discuss the results of the open card sort.

4.2.1 Demographics

In total, 176 software developers filled out the entire survey over a period of seven weeks. The vast majority of them have at least three years of experience in software development: less than 1 year (2.8%), 1-2 years (6.8%), 3-6 years (31.8%), 7-10 years (21.6%) and more than 10 years (36.9%). Regarding the languages used by most respondents: 84.1% indicated that they use Java, followed by 55.1% for JavaScript and 39.2% for Python. The languages

PHP, C, C++ and C# were all selected by around 25% of the developers. The least selected languages are R (16.5%), Swift (6.3%) and Objective-C (5.1%). Finally, 44 developers indicated the use of another, in total 24 different languages, of which Scala (11) is the most mentioned, followed by Ruby (8). The most selected IDE is Eclipse (31.8%), followed by IntelliJ (30.7%) and Visual Studio (11.9%).

4.2.2 Description and interpretation of survey answers

Next, we asked the developers whether or not they use their IDE debugger. 143 developers (81.3%) indicate that they use the IDE-provided debugging infrastructure, 15 (8.5%) indicate they do not use it and 18 developers (10.2%) indicate that their selected IDE does not have a debugger. The most selected answer besides using the IDE debugger that the respondents indicate they use for debugging is examining log files (72.2%), followed closely by using print statements (71.6%). Other answers included the usage of an external program (21.0%) or additional other techniques (30.1%). In addition, 19 developers indicate the use of another method, of which adding or running tests and using web development tools built in to the browser were mentioned most (both four times).

> Most software developers claim to be using the IDE-provided debugging infrastructure in conjunction with examining log files and using print statements.

Furthermore, of the 15 developers not using the debugging infrastructure, 8 think that print statements are more effective/efficient, 6 find techniques other than print statements more effective/efficient, 6 use an external program they find more effective/efficient, 4 do not know how to use the debugger and one thinks his program is impossible to debug.

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

The 143 developers that indicated that they use the IDE-provided debugging infrastructure were asked more detailed questions on whether or not they know and use specific debugging features. The results for these questions are shown in Figures 4.1 to 4.3 for the questions on breakpoint types, breakpoint options and other debugging features, respectively. Regarding the questions on the usage of (unit) tests throughout the debugging process, Figure 4.4 visualizes the results based on all 176 responses.

Figure 4.1 shows that most developers are familiar with line, exception, method and field breakpoints, while temporary line breakpoints and class prepare breakpoints are known by much less developers. In addition, while line breakpoints are also used by the vast majority of developers, other types of breakpoints are used by less than half of the respondents to even by almost none of them.

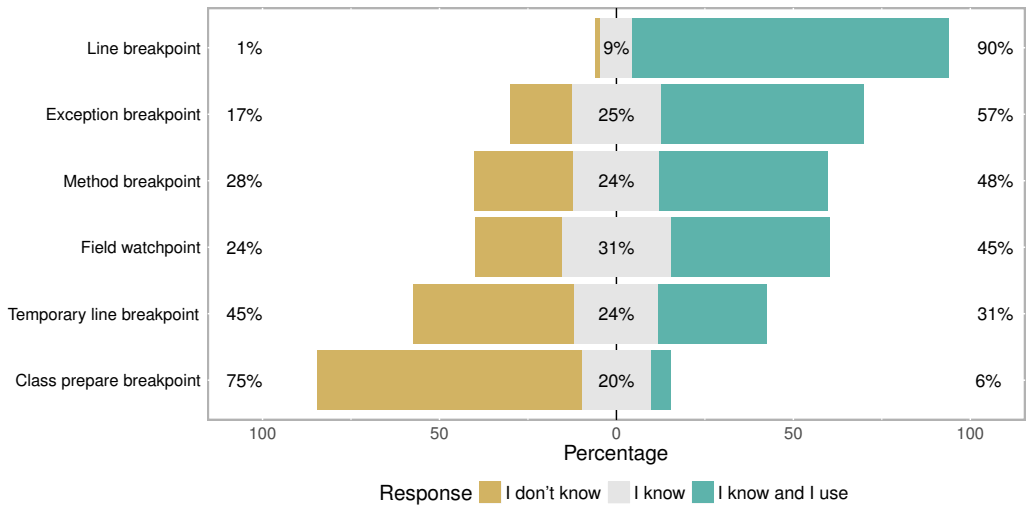

Figure 4.1: Distribution of answers in the questions on breakpoint types ($n = 143$).

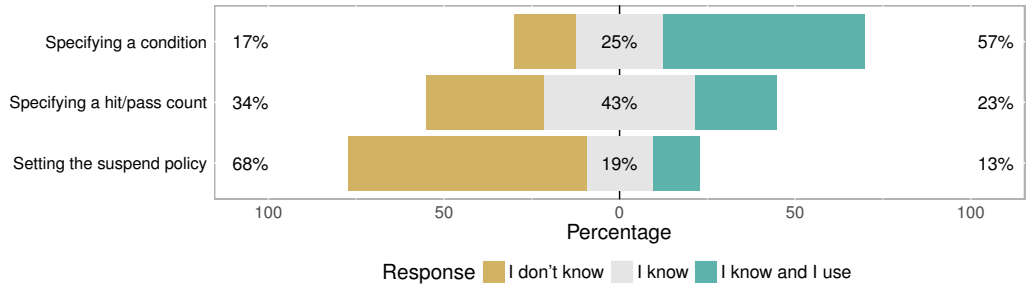

Figure 4.2: Distribution of answers in the questions on breakpoint options ($n = 143$).

Line breakpoints are used by the vast majority of developers, while other, more advanced types are not and are even unknown to many developers.

Figure 4.2 indicates that the majority of developers specify conditions on the breakpoint to only let the program execution be suspended if the condition is true. However, specifying a hit count that makes sure the program execution is only suspended after the line is executed the specified number of times, or setting a suspend policy that indicates whether the entire program or just one thread needs to be suspended once the breakpoint is hit, are both known less and used.

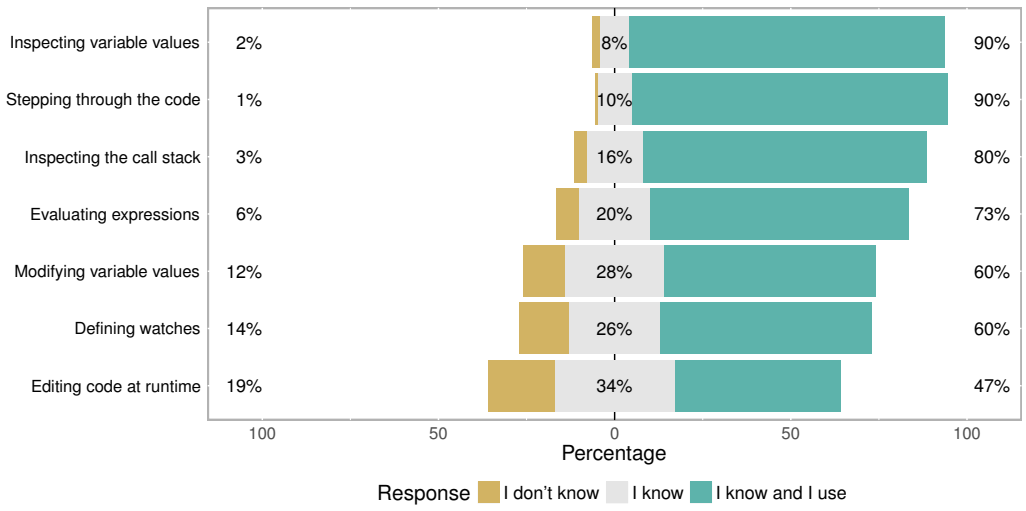

Figure 4.3: Distribution of answers in the questions on debugging features ($n = 143$).

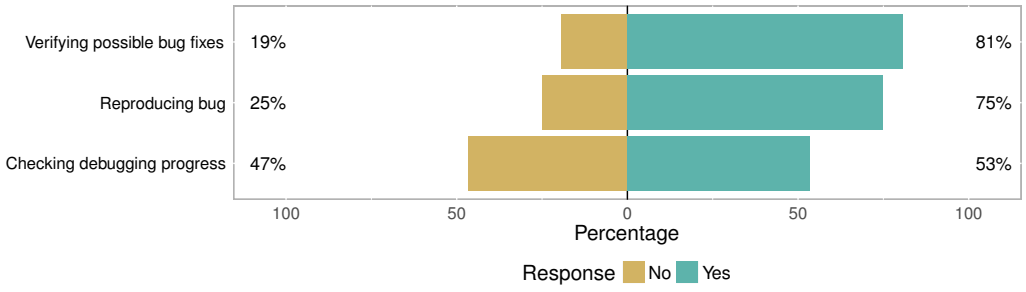

Figure 4.4: Distribution of answers in the questions on unit tests ($n = 176$).

Most developers claim to be familiar with breakpoint conditions, but not with the hit count and suspend policy.

The results in Figure 4.3 show that over 80% of the developers seem to know all major debugging features found in most modern IDEs. However, the more advanced the features get, like e.g. defining watches or editing code at runtime, the less they seem to be used according to the respondents.

Although most developers claim to known all common debugging features, these features are used less as they get more advanced.

Looking at the responses for the (unit) testing questions, Figure 4.4 indicates that tests are often used at the start and end of the debugging process, for reproducing bugs and verifying bug fixes, respectively, and a bit less throughout the process.

> Developers claim that testing is an integral part of the debugging process, especially at the beginning and end.

4.2.3 Dependency analysis of survey responses

To examine the relationship between the categorical survey answers, we first computed a pair-wise dependency test and then, if found to be dependent, the strength of their relationship. For example, this allows us to understand whether and how strongly programmer experience correlates with the use of debugger features like breakpoints, watches or the use of testing to guide debugging.

For the dependency test between survey answers, we used a *Pearson Chi-Squared test of independence* as we are dealing with categorical variables. In order to be able to compute the strength of their relationship with a *Spearman rank-order correlation test*, we had to convert each categorical answer to an ordinal scale using a linear integer transformation describing its rank. This was sound because our predefined answer options have a naturally ranked order (e.g., "I don't know" = 1, "I know" = 2, "I know and use" = 3). Figure 4.5 shows the correlation strength ρ on all $n = 176$ survey responses and Figure 4.6 visualizes the strength on all $n = 143$ respondents that indicated to use the debugger. These figures allow for an intuitive overview of the relations. The more intense the color and the flatter the ellipses are above its diagonal, the higher is the strength of the correlation, which is reported numerically below the diagonal. Empty cells correspond to non-significant results of the chi-squared (χ^2) test at a 95% confidence interval ($\alpha = 0.05$).

For n independent cross-correlations we perform on a given significance level α, there is a $1-(1-\alpha)^n$ likelihood for at least one relationship to have occurred by chance. Plugging in our standard $\alpha = 0.05$ and 110 correlation pairs from Figure 4.6, we receive a 99.6% likelihood of at least one acclaimed dependency according to our χ test that does not hold in reality. While this does not affect the general validity of our results, one must recheck the results through additional sources or mixed methods when reasoning about a single relationship from Figure 4.6.

Based on the results in Figures 4.5 and 4.6 and Hopkin's guidelines described above, we find that there is no correlation between the usage of the IDE-provided debugging infrastructure or (unit) tests for debugging and experience in software development. However, there is a weak correlation between experience and specifying hit counts and a moderate correlation between experience and the usage of watches during debugging.

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

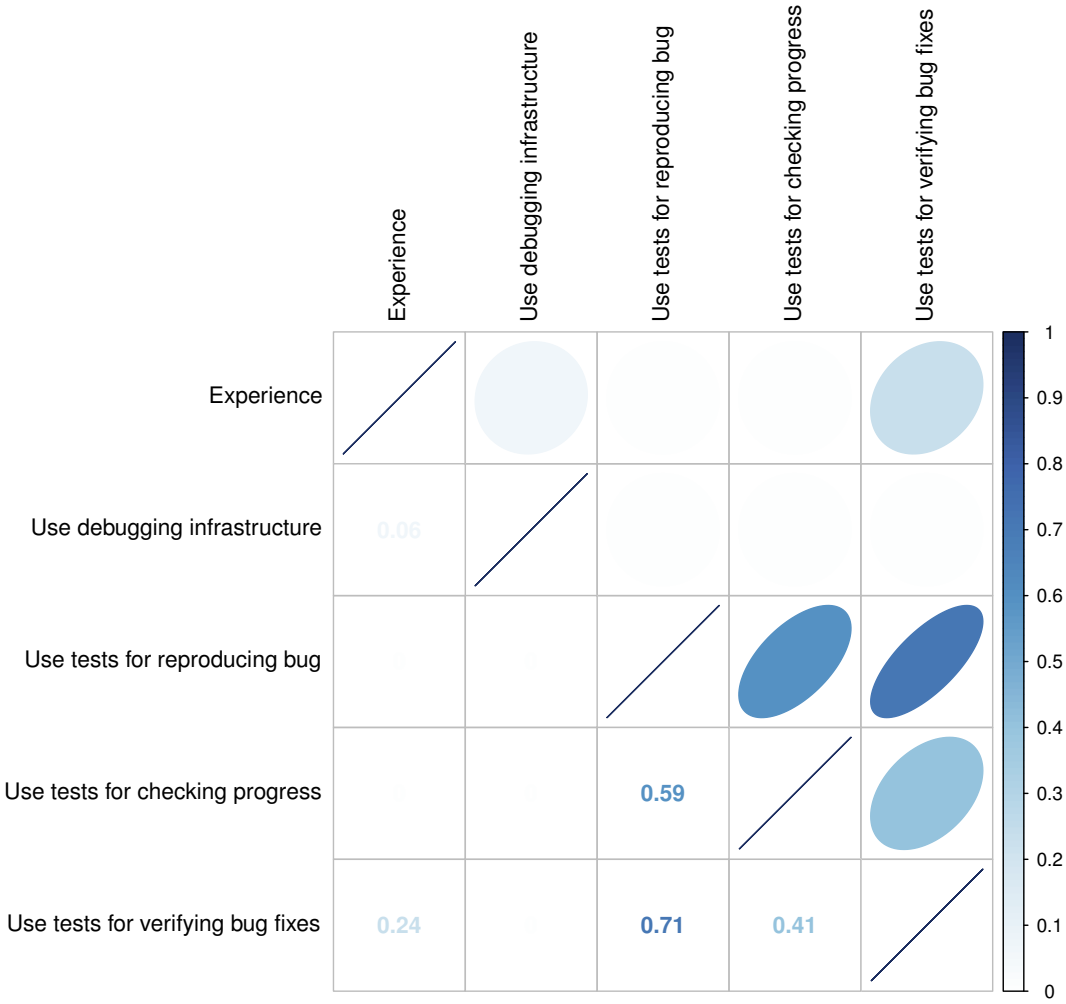

Figure 4.5: Correlation analysis using Spearman rank correlation between several questions with all responses ($n = 176$).

In addition, we find that there is a moderate correlation between the usage of tests at the beginning and end of the debugging process to reproduce bugs and verify bug fixes, respectively, and a weak to moderate correlation between using tests at the beginning or end and throughout the process for checking progress.

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

Figure 4.6 shows many more examples of weak and moderate correlation between the knowledge and usage of several debugging features. One thing to note is that there is quite

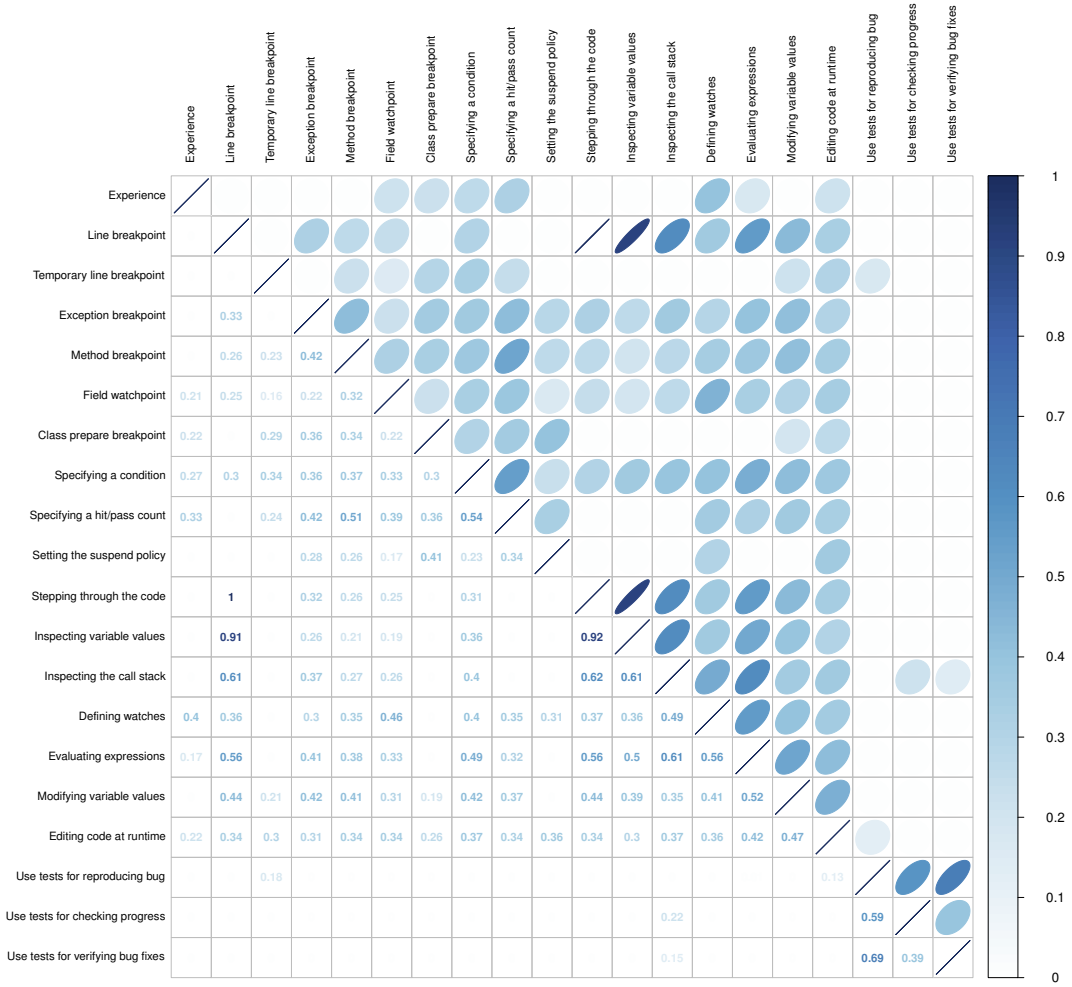

Figure 4.6: Correlation analysis using Spearman rank correlation between several questions with 'uses debugger' responses ($n = 143$).

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

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

4.2.4 Results of open card sorting

In total, 108 respondents gave a response to the statement that "the best invention in debugging still was printf debugging". After performing the open card sorting as described above, we identified 34 different tags, of which the name, frequency and (optional) description are

Name	Frequency	Description
advanced debugging techniques	5	Use more advanced techniques than printf debugging
agree	20	
avoid debugging	6	
before debugger	4	Use some techniques before using the debugger
combination of techniques	19	Use multiple techniques for debugging
concurrency	4	Concurrent environment
crash dump	1	Use a crash dump for debugging
debugger jittery	3	Debugger interferes with thinking process
debugger overhead	6	Debugger has (too much) impact on performance
debuggers better	30	
debuggers interference	4	Debugger interferes with program execution
depends	8	Used technique depends on bug or situation
disagree	31	
easy	11	Technique is easy to apply
embedded	1	Embedded systems environment
external libraries	1	Environment with external libraries
fast	12	Technique is fast
first printf	6	First use printf, then other techniques if necessary
great web dev tools	3	Use web developer tools for debugging
hard problems	4	Use debugger for hard(er) problems
live IDE	4	Use a live IDE instead of debugging
logging	8	Use logging for debugging
outdated	3	Printf is outdated
partial agree	27	
printf not enough	10	
printf not good practice	7	
printf when no debugger	4	
profiler	1	Use a profiler for debugging
program state	6	Debugger allows for inspecting the program state
remote	7	Remote environment
REPL	3	Use a Read-Execute-Print Loop for debugging
testing better	7	Use tests instead of other techniques
time-consuming	3	Technique is time-consuming to apply
uses printf	51	Use printf for debugging

Table 4.1: Resulting tags with their frequency and (optional) description.

given in Table 4.1. To see which tags often co-occur and therefore to see which arguments are commonly given by developers, if any, we analysed the co-occurrences between all pairs of tags. To visualize these relations, we decided to use a graph representation in which the vertices correspond to the tags and the undirected, weighted edges to the co-occurrences of two tags. The size of the vertices in this graph is determined by the occurrence frequency of the tag, while the weight of the edges is determined by the number of co-occurrences. Figure 4.7 shows the graph resulting from this process.

However, in this representation, vertices with a relatively low occurrence frequency can never have an edge with a relatively high weight. Nevertheless, these vertices might actually

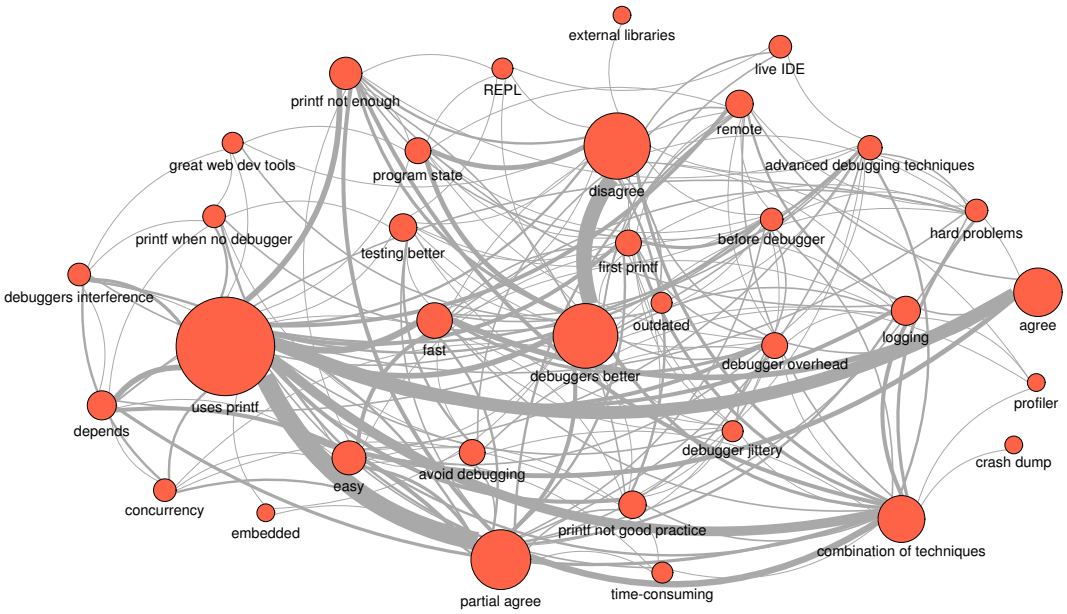

Figure 4.7: Occurrence frequency and relationship of all tags extracted from survey answers to the open question.

have a very strong connection with other tags, which means that if the tag is listed, it often co-occurs with one or more other tags. To improve the visualization, this shortcoming was fixed by normalizing the weights of the edges using the following formula:

$$w_{norm} = \frac{w}{\max\left(f_{source}, f_{destination}\right)} \tag{4.1}$$

where w_{norm} is the normalized edge weight, w the original edge weight and f_{source} and $f_{destination}$ the occurrence frequency of the source and destination tag, respectively. The result of this normalization is shown in Figure 4.8.

As the graph in Figure 4.8 is still too dense to clearly see all connections, we filtered out all edges for which $w_{norm} < 0.25$ and removed the vertices that did not have any outgoing or incoming edges after this filtering process. The resulting graph, in which only the relatively strong connections between tags remain, is shown in Figure 4.9

Based on the connected subgraphs in Figure 4.9, which indicate that some tags are often mentioned together in answers to the open question, we can identify (at least) the following different types of responses:

- Debuggers are not suitable for finding bugs in a concurrent environment as they interfere too much.
- Avoid debugging by using tests instead.
- Using IDE debuggers is better than using print statements.

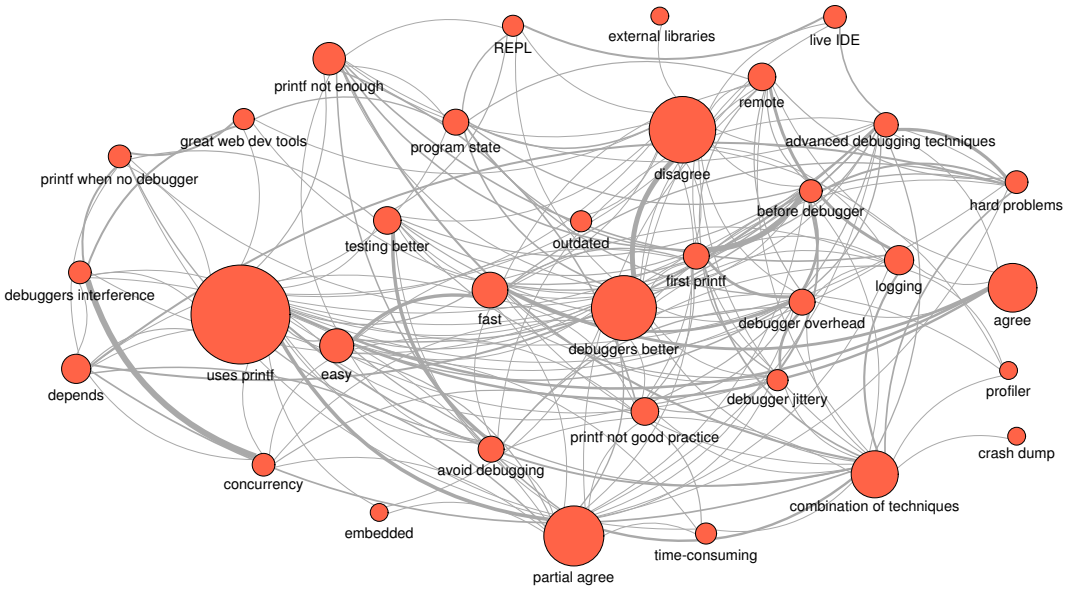

Figure 4.8: Occurrence frequency and normalized relationship of all tags extracted from survey answers to the open question.

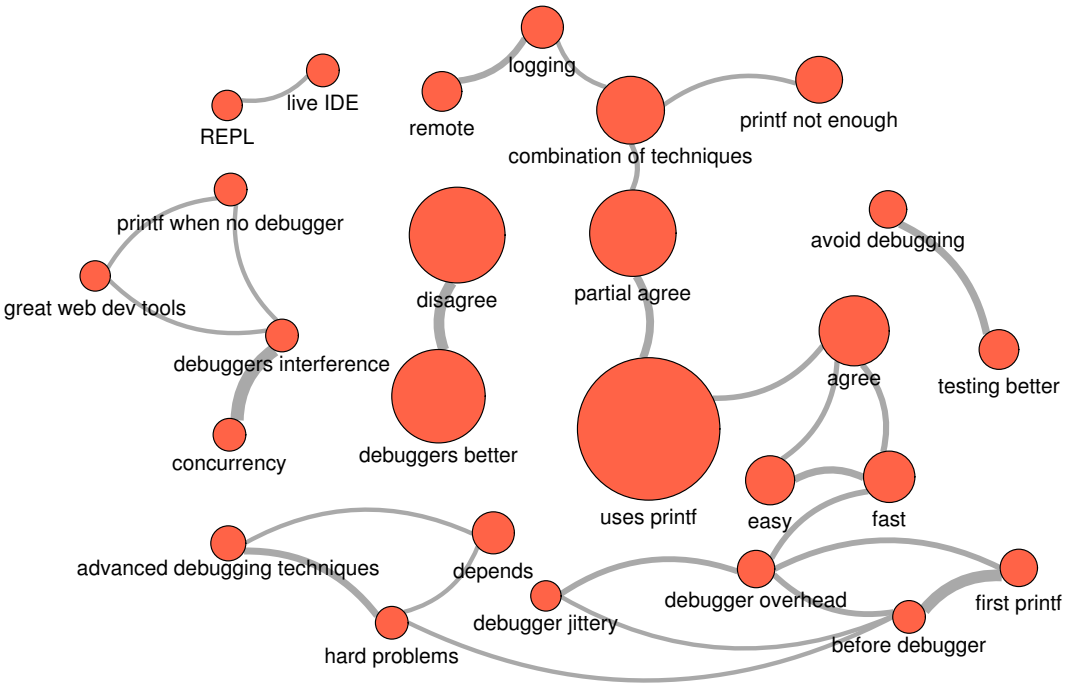

Figure 4.9: Occurrence frequencies and normalized, strong, relationships of some tags extracted from survey answers to the open question.

- Use logging for debugging remote applications.
- Printf debugging is not enough, a combination of techniques is required.
- A live IDE with a REPL is better than using print statements.
- First use print statements for debugging, then use a debugger or other advanced debugging techniques for harder bugs.
- Printf debugging is the best for debugging as it is fast and easy.
- Debuggers can have (too much) overhead and are jittery.

Chapter 5

Tracking Debugging Behavior with WatchDog 2.0

This chapter on extending WatchDog to WatchDog 2.0 is structured as follows. First, Section 5.1 will give an overview of WatchDog's functionality and design at the start of this study. Then, Section 5.2 will present WatchDog 2.0's new features and architecture at the end of the study. Finally, Section 5.3 will describe the process of how we added these new features.

5.1 Existing functionality and architecture

In this section we first discuss the purpose of WatchDog and its main features. Then, we take a closer look at its internals by describing its software architecture.

5.1.1 Purpose and functionality

WatchDog, as developed by Beller et al., is "a research vehicle that tracks the testing habits of developers" [6]. In particular, it is a plugin for Eclipse and IntelliJ that records all developer behavior related to reading and writing production code as well as reading, writing and executing test code. Their purpose for developing these plugins is to understand "how developers test and how to better support them in practice" by collecting "usage data related to developer testing" on a larger scale than what is possible by looking over their shoulders [5].

To incentivize developers to install WatchDog, it also provides its users with "immediate testing and development analytics" [6]. In particular, developers can open the so-called 'WatchDog View' that provides them with some basic statistics on their development behavior. An example of what this view might look like is given in Figure 5.1. In addition, users can also request a report that contains more detailed statistics on their development behavior for their registered project. An example of such a (partial) report is shown in Figure 5.2. Not only the developer's own statistics are shown, but also the mean statistics, which enables developers to compare themselves with others.

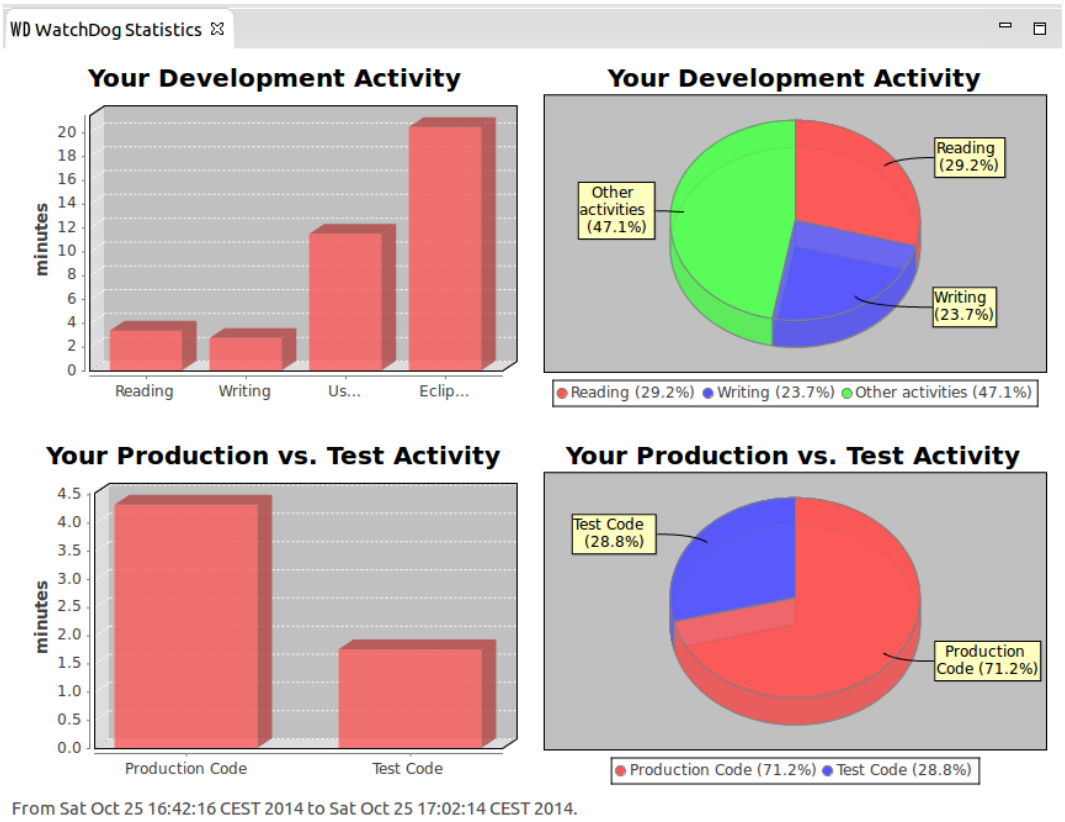

Figure 5.1: An example of the 'WatchDog View' in Eclipse (source: [49]).

5.1.2 Global architecture

As explained in [6], WatchDog has a 3-layer architecture that is visualized in Figure 5.3. The figure reveals the three layers that make up WatchDog: the client, the server and the analytics pipeline.

The client is the actual plugin that lives inside the IDE. As plugins need to be tailored to the host IDE, a separate client has to be implemented for each supported IDE. Currently, separate clients are available for both Eclipse and IntelliJ. To avoid having lots of duplicate code, all clients share are common core that contains all IDE-independent functionality as well as specifications for functionality that should be implemented by all clients.

The server is the entity that controls the storage of intervals, users and projects in a Mongo database. These three concepts form the heart of WatchDog as intervals capture the actual development behavior, while users and projects are used to link intervals to individual developers or software projects. Whenever a new interval is closed, e.g. the user stopped reading code and started writing, it is first recorded in a local database. Then, at a regular rate, all contents of this database are transferred to the remote server in order to permanently store the recorded intervals.

To give some more insights into how WatchDog is structured and functioning internally,

Summary of your Test-Driven Development Practices

You followed Test-Driven Development (TDD) 7.13% of your development changes (so, in words, seldom). With this TDD followship, your project is in the top 54 (1.76%) of all WatchDog projects. Your TDD cycle is made up of 96.86% refactoring and 3.14% testing phase.

Detailed Statistics

In the following table, you can find more detailed statistics on your project.

Description	Your value	Mean
Total time in which WatchDog was active	151.9h	78h
Time averaged per day	1h / day	4.9h / day

General Development Behavior	Your value	Mean
Active Eclipse Usage (of the time Eclipse was open)	100%	40%
Time spent Writing	11%	30%
Time spent Reading	8%	30%

Java Development Behaviour	Your value	Mean
Time spent writing Java code	43%	49%
Time spent reading Java code	57%	49%
Time spent in debug mode	0% (15h)	2h

Testing Behaviour	Your value	Mean
Estimated Time Working on Tests	15%	73%
Actual time working on testing	14%	10%
Estimated Time Working on Production	85%	26%
Actual time spent on production code	86%	89%

Test Execution Behaviour	Your value	Mean
Number of test executions	184	23
Number of test executions per day	2/day	2.17/day
Number of failing tests	80 (43%)	13.23 (58%)
Average test run duration	14.45 sec	3.82 sec

Figure 5.2: An example of a (partial) WatchDog report (source: [49]).

Figure 5.4 shows a package diagram of the main packages that form the client, both in the common core and the IDE-specific projects. This diagram clearly indicates that all three parts have similar package names. This is because the common core often contains only specifications or basic implementations of certain entities that can only be (further) implemented using the IDE-specific functionality. Using the same package names then makes it much clearer what parts are related to each other.

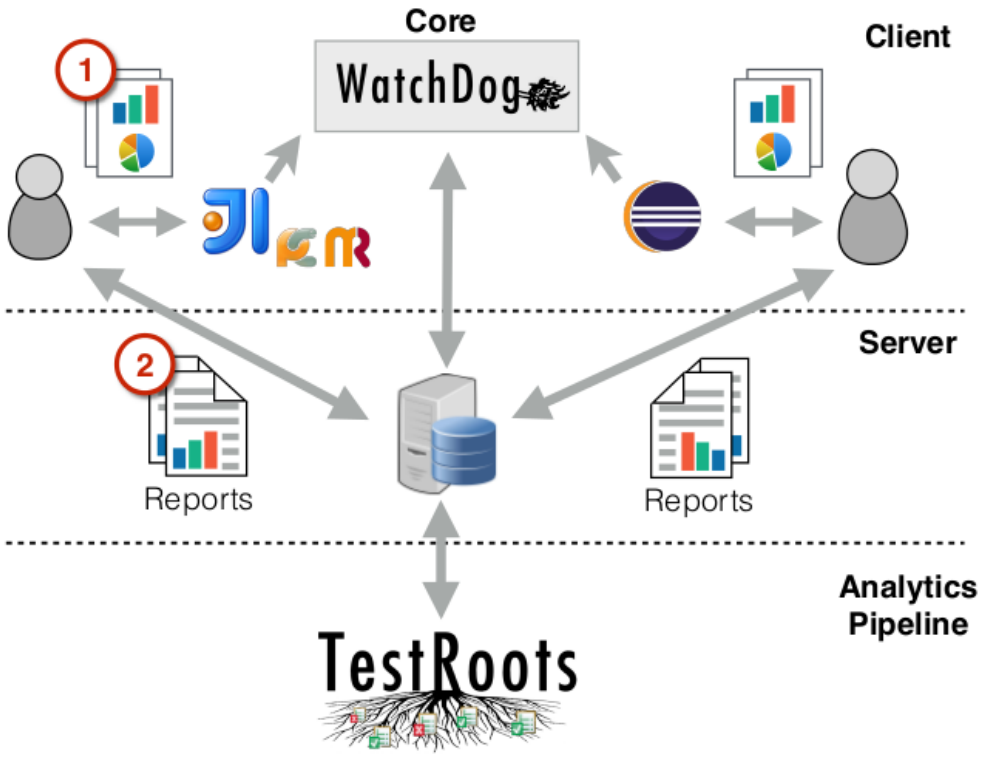

Figure 5.3: WatchDog's three layer architecture (source: [6]).

5.2 New functionality and architecture

This section first explains the differences in purpose and functionality between WatchDog and WatchDog 2.0. Then, we will describe the differences between their architectural designs as well.

5.2.1 New functionality

The main difference in terms of functionality between WatchDog and WatchDog 2.0 is that WatchDog 2.0 also tracks the developer's debugging behavior in addition to its testing behavior. In particular, WatchDog 2.0 records when and for how long the user launches the debugger. In addition, it tracks which actions the user performs within and between these debugging sessions with regards to breakpoints and debugging features. To get an idea of which specific actions we track with WatchDog 2.0, Figures 5.5 to 5.7 highlight some of these actions in the breakpoint, debug and editor window of IntelliJ, respectively. To see which actions belong to the numbers shown these figures, see Table 5.1. Similar actions exist in Eclipse, which we also track in WatchDog 2.0. As the figures show, multiple ways of performing the same actions are tracked.

To create an incentive for users to update to or install WatchDog 2.0 in the first place,

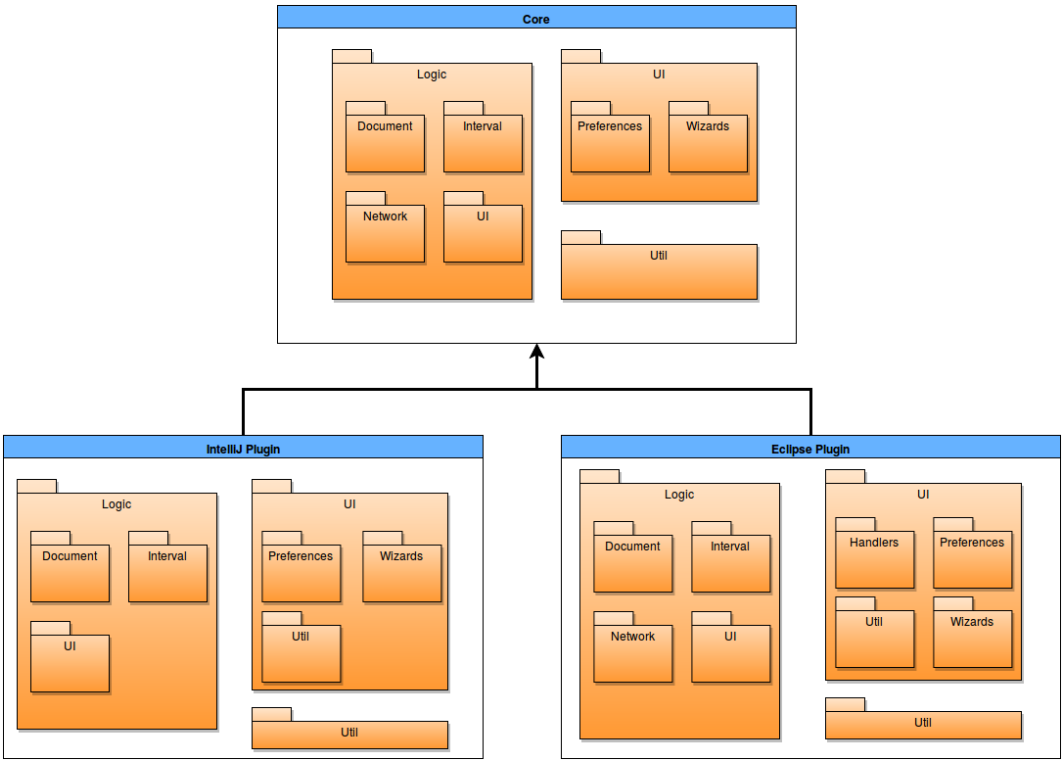

Figure 5.4: Packages in WatchDog's existing architecture.

we decided to update both the 'WatchDog View' and the project reports that are generated on a daily basis. For the project reports we included three new statistics: the number of debugging sessions the user performed as well as their total and average duration. Just like in the existing reports, the mean values over all projects are also shown to let users compare their debugging behavior with others. An example of what this new section in the reports look like can be found in Figure 5.8.

For updating the 'WatchDog View', we decided to add the following two components to the existing view: some basic debugging statistics and a Gantt chart showing the events that occurred during the selected debugging interval. In particular, the following debugging statistics are shown to the user: the number of debugging intervals in the selected time period, the total amount of time spent in the debugger as well as its percentage of the active IDE time and the average length of a debugging session in seconds. The Gantt chart visualizes each event that occurred during the selected debug interval on a timeline in such a way that a user can see what he or she has actually done during a debugging session. The debug interval of which the events should be visualized can be selected from a dropdown that shows the ten latest debug intervals. Finally, the view was also separated in three parts called 'General', 'Testing' and 'Debugging' in order to create a clear separation between the purpose of the different charts. An example of the debugging part that was added in WatchDog 2.0 can be seen in Figure 5.9.

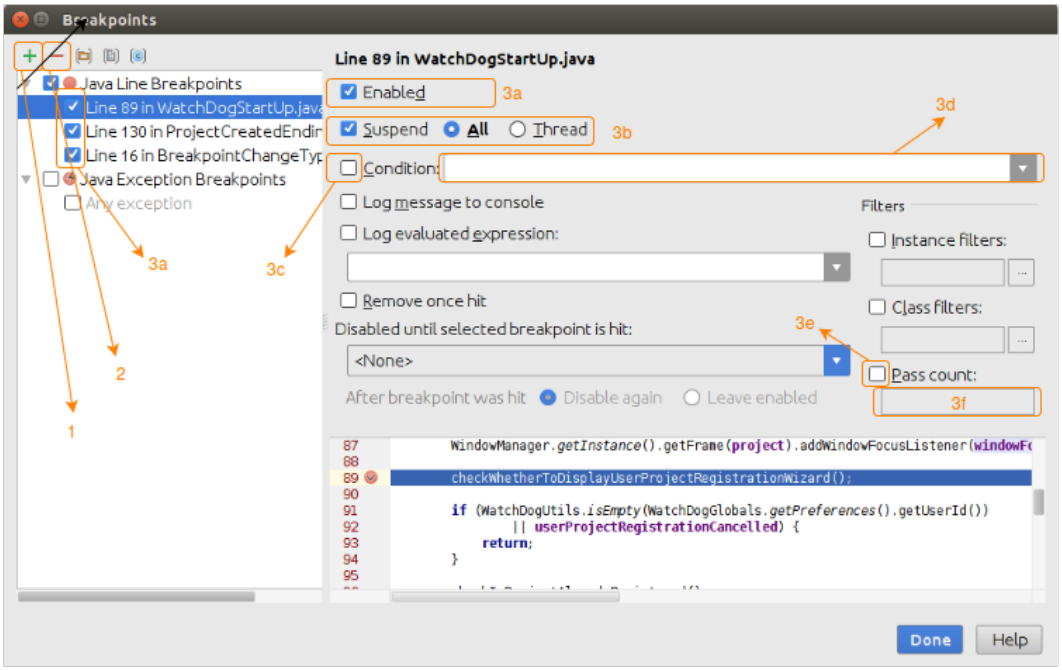

Figure 5.5: Breakpoint window in IntelliJ with actions that WatchDog 2.0 tracks highlighted.

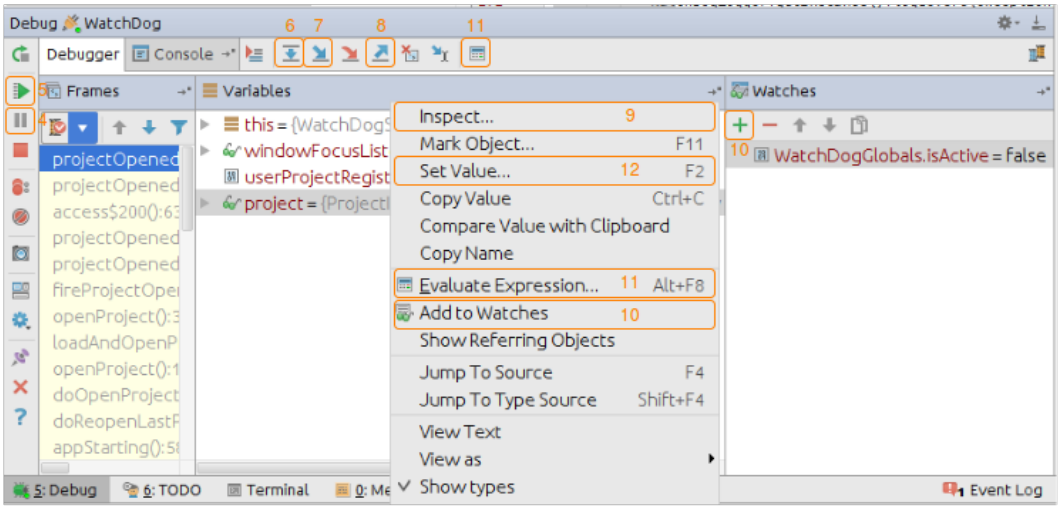

Figure 5.6: Debug window in IntelliJ with actions that WatchDog 2.0 tracks highlighted.

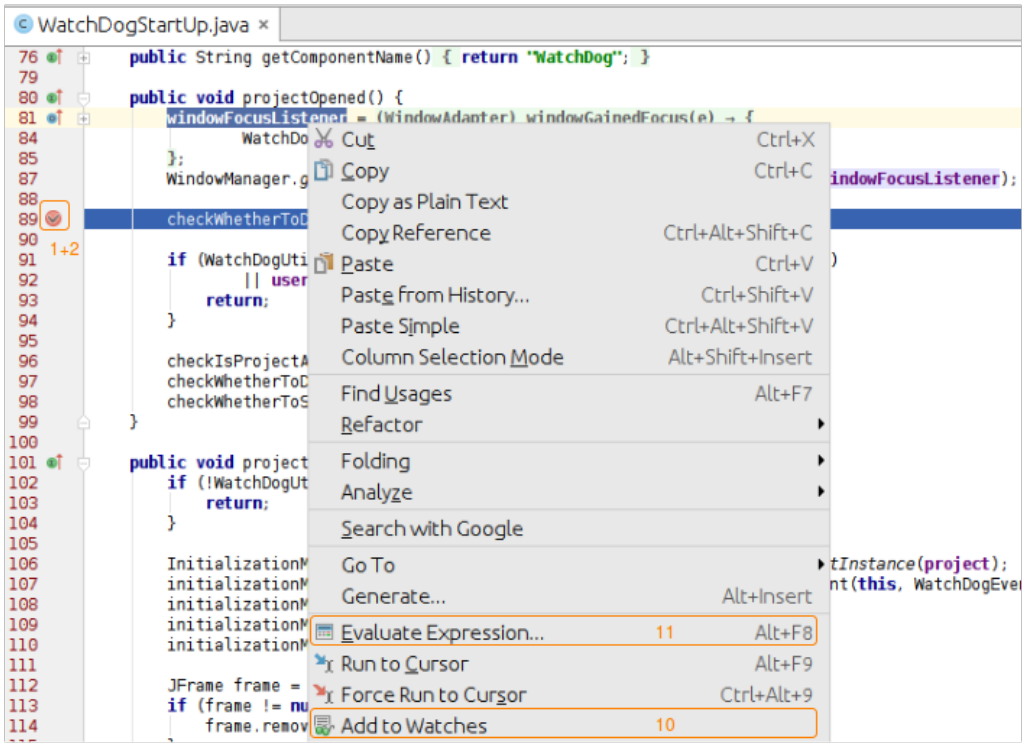

Figure 5.7: Editor window in IntelliJ with actions that WatchDog 2.0 tracks highlighted.

Debugging Behavior	Your value	Mean
Number of debugging sessions	15	1
Total duration of debugging sessions	1029.4 seconds	488 seconds
Average debugging session duration	68.6 seconds	928.6 seconds

Figure 5.8: Example of the new debugging section within WatchDog 2.0's project reports.

5.2.2 New architecture

The overall 3-layer architecture of WatchDog is still in place in WatchDog 2.0. However, both the common core as well as IDE-specific plugins were significantly adapted and extended. To get an impression of how the project structure changed, Figure 5.10 shows the package diagram of WatchDog 2.0, in which the new packages are highlighted. In this way, the main architectural changes between WatchDog and WatchDog 2.0 become apparent.

As the package diagram shows, we captured the tracking of the actions shown in Table 5.1 in the concept of an *event*. These events are generated whenever a developer clicks a button or presses the key(s) belonging to a particular debugging or breakpoint action and contain the type of the event, its timestamp and some optional other properties like e.g. the breakpoint type for 'Add breakpoint' events.

Number	Action(s)
1	Add breakpoint
2	Remove breakpoint
3a	Enable or disable breakpoint
3b	Set suspend policy of breakpoint
3c	Enable or disable the breakpoint condition
3d	Change the breakpoint condition
3e	Add or remove the hit count of a breakpoint
3f	Change the hit count of a breakpoint
4	Suspend the program execution
5	Resume the program execution
6	Step over the current line
7	Step into the current line
8	Step out of the current frame
9	Inspect a variable
10	Define a watch
11	Evaluate an expression
12	Modify a variable value

Table 5.1: Actions belonging to the numbers in Figures 5.5 to 5.7.

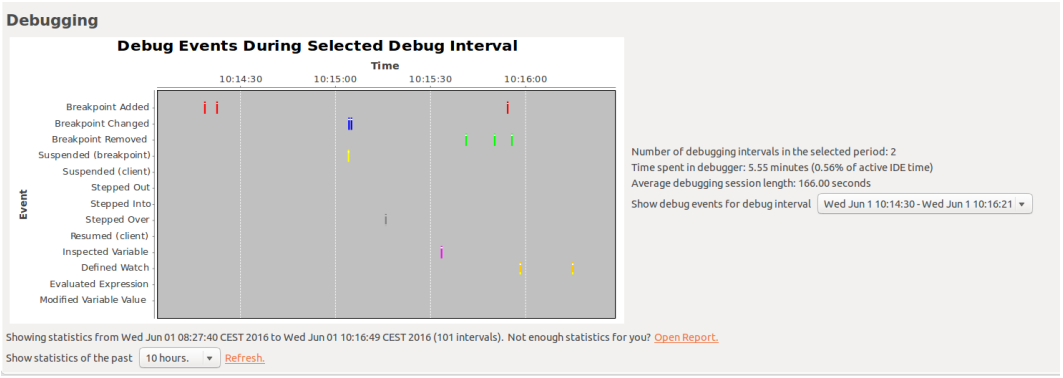

Figure 5.9: New ‘debugging part’ within the ‘WatchDog View’ of WatchDog 2.0.

5.3 Development process

In this section we give some more insights into the process of transforming WatchDog to WatchDog 2.0. In particular, we discuss the refactoring performed at the beginning of the project followed by its implementation and testing process.

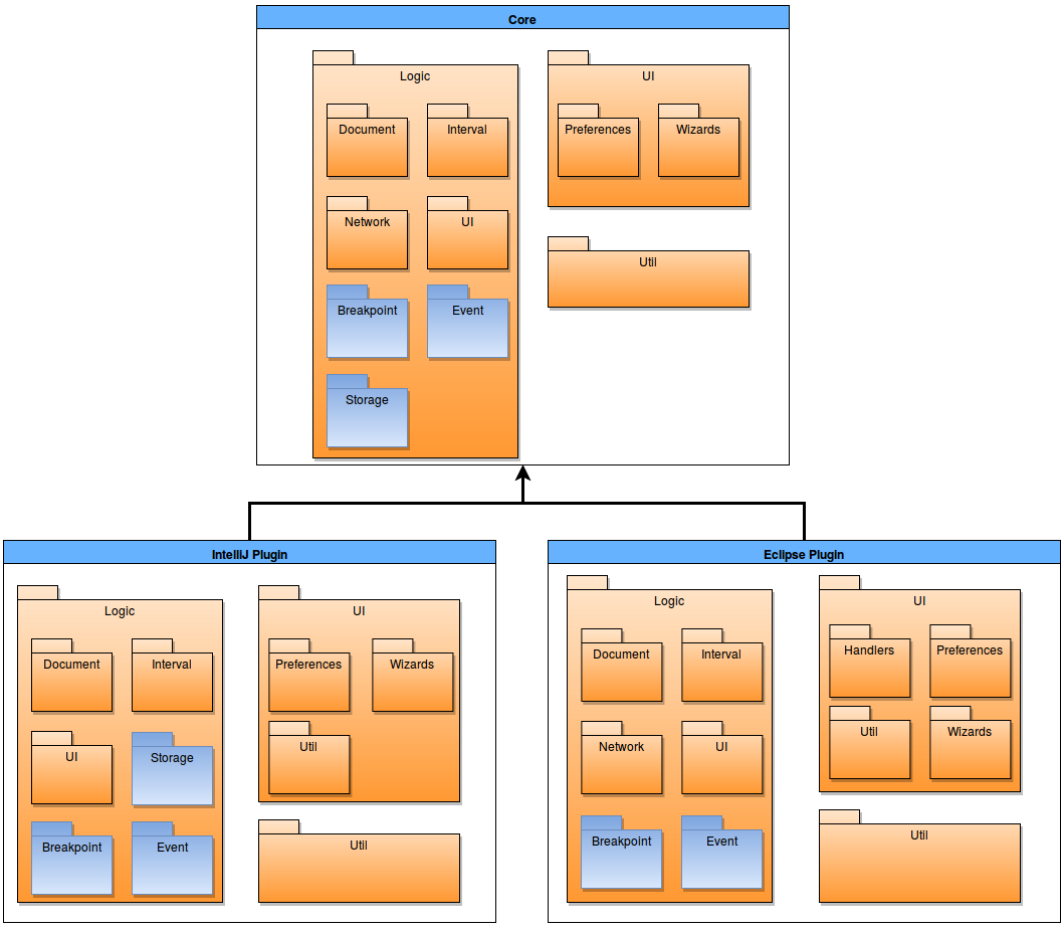

Figure 5.10: Packages in WatchDog 2.0's new architecture.

5.3.1 Refactoring

As described above, WatchDog's architecture has a common core that is shared by both clients in order to alleviate "the maintenance difficulties of two forks" [6]. However, while familiarizing with the codebase in order to be able to extend it later on, we noticed that still quite some duplicate code existed between the Eclipse and IntelliJ project. Having duplicate code should be avoided according to the Don't Repeat Yourself (DRY) principle [17]. Furthermore, duplicate code is also one of the code smells that should be refactored according to Fowler et al [13].

Therefore, by following Uncle Bob's Boy Scout Rule of leaving the code in a better state than you found it [25], the existing architecture was refactored. In particular, quite some duplicate code was refactored away by pulling it into the common core. To be able to locate the duplicate code the PMD Copy/Paste Detector (CPD) [18] was used. The result of this refactoring was a significant drop in the clone coverage percentage as reported by Teamscale [15]. The clone coverage trend belonging to WatchDog is visualized in Figure 5.11.

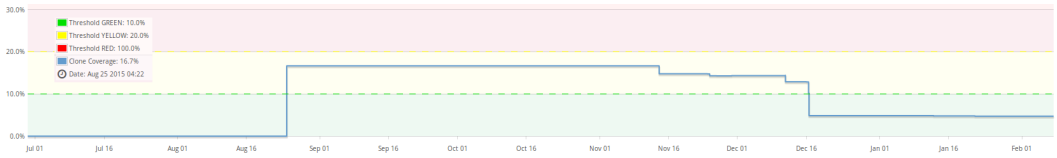

Figure 5.11: WatchDog's clone coverage trend as reported by Teamscale.

In this figure you can see that the clone percentage became 16.7% on August 24, which is the moment when support for IntelliJ was added to the codebase. Before the refactorings that started on December 11, the clone coverage percentage was 14.3%, which is still quite high. As can be seen by the dropping line in the figure around December 16, the refactorings significantly reduced the clone coverage to an acceptable 4.9%.

5.3.2 Implementation process

To apply structure to the development process of WatchDog 2.0, it was split up into four phases based on the research question they are supposed to (partially) answer. In this way, we created several different smaller tasks instead of one big project, which also makes the process more manageable. The four resulting phases that are required to answer the two main research questions were:

1. Add support for tracking debug intervals (for **RQ2**).
2. Add support for tracking breakpoint events (for **RQ3**).
3. Add support for tracking debugging events (for **RQ3**).
4. Visualize the new data in the 'WatchDog View' and the project report.

The first phase, of which the changes can be found in Pull Request 235[1], was the simplest to complete and also did not require any architectural changes. The second phase contained the largest changes as it required the creation of the functionality for creating, storing and transferring events. Furthermore, this phase required quite some research to get to know which type of breakpoints exist in both Eclipse and IntelliJ and which events we can listen to using the IDE-specific functions and how. The changes we made for this phase can be found in Pull Request 236[2]. At the start of phase three, we again performed extensive research to get to know which debugging features are supported and can also be listened to in both Eclipse and IntelliJ. Support for tracking these events was added by the changes made in Pull Request 239[3]. Changes we made for phase four can be found in Pull Request

[1] https://github.com/TestRoots/watchdog/pull/235
[2] https://github.com/TestRoots/watchdog/pull/236
[3] https://github.com/TestRoots/watchdog/pull/239

242[4]. During the implementation process, we improved and refactored the new functionality several times in order to improve both WatchDog 2.0's internal as external quality. An example of such improvements can be found in Pull Request 253[5].

5.3.3 Testing

To ensure the new functionality described above works as expected and that the existing functionality was not changed throughout the process, several testing procedures were in place. First of all, the existing codebase already contained quite some JUnit tests, also ones that use Mockito[6] to mock external dependencies. Therefore, we used these tests for regression testing. In addition, WatchDog's development process ensures that the main branch is always working by using Travis CI[7] for continuous integration. In this way, code changes are only merged into the main branch when this build is marked as successful, which means that all binaries can be build and that all tests pass. Furthermore, several static analysis tools are in place to monitor the internal quality of the code base.

For testing the new functionality itself, we added unit tests during the phases of the development process. For instance, before adding the events endpoint to the server in phase two, we first extended the RSpec tests to also test for this endpoint. Then, we added the actual functionality until all of these tests passed. Also during the development of the client, we implemented new unit tests along the way. In this way, the new functionality was not only tested once, but we also included these new tests for regression testing.

Besides these automated testing techniques, we also relied on quite some manual testing during the development of the debug event tracking. This was necessary because the listeners for such events cannot (easily) be tested using automated techniques as actual user interactions are required to fire the events. Therefore, we performed these interactions manually and then we checked the generation of the actual events in WatchDog 2.0 using either log statements or by setting breakpoints inside the listeners in order to inspect more complex events. For the similar reasons the extension of the IDE-specific parts of the new 'WatchDog View' were also tested manually.

[4] https://github.com/TestRoots/watchdog/pull/242
[5] https://github.com/TestRoots/watchdog/pull/253
[6] http://mockito.org/
[7] https://travis-ci.org/

Chapter 6

Analysing Debugging Behavior with WatchDog 2.0

To get to know how developers debug in their IDE, this chapter first describes how we used WatchDog 2.0 to collect the required behavioral data. Then, the results we found in the collected data will be described and interpreted.

6.1 Research design and methodology

This section first discusses how we collected the data using WatchDog 2.0. Next, we describe the analysis methods used for answering the sub research questions we described in the introduction.

6.1.1 Data collection

To collect as much data on the debugging behavior of developers as possible, we tried to attract as many WatchDog 2.0 users as possible. Therefore, we first rebranded WatchDog to WatchDog 2.0 by updating both the project's website[1] and the plugin descriptions on the Eclipse MarketPlace[2] and IntelliJ Plugin Repository[3]. In this way, we informed potential new WatchDog 2.0 users on the new functionality for tracking debugging behavior.

As WatchDog already has a relatively large userbase, we also tried to convince those users to update their installation to WatchDog 2.0. For this, we used the existing functionality in WatchDog to inform users of a major update to the plugin. In this way, we showed all users a dialog that informs them of the update and provides them with a button to immediately apply this update.

[1] https://testroots.org/

[2] http://marketplace.eclipse.org/content/testroots-watchdog

[3] https://plugins.jetbrains.com/plugin/7828

6.1.2 Data analysis methods

To analyse the data collected with WatchDog 2.0, we integrated the new required debugging analyses into the existing analysis pipeline of WatchDog. This pipeline basically extracts the data from the Mongo database and stores all necessary information in CSV files using R. Then, this data can be loaded back into R and analysed, which is exactly what we did for the analyses required for answering the sub research questions. The analysis methods we used for some of these research questions require some more explanation.

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

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

Finally, for **RQ2.4**, **RQ2.6** and **RQ2.7** we again used the non-parametric Spearman rank-order correlation test for analysing dependencies. To interpret the results of these dependency analyses, we again use Hopkins' guidelines [16] as described in the analysis methods used for the online survey.

6.2 WatchDog 2.0 results and their interpretations

In this section we first describe the demographics of the observed data and extract some basic results from it. Then, we describe the results and their interpretations per sub research question.

6.2.1 Demographics and basic results

Since the release of WatchDog 2.0 on 22 April 2016, we collected user data for a period over two months, until 28 June 2016. In this period, we received 1,155,189 intervals, from 458 users in 603 projects. Of these, 3,142 were debug intervals from 132 developers. In total, we recorded 18,156 hours in which the IDE was open, which amounts to 10.3 observed developer years based on the average of 1,770 working hours per year for OECD countries[4]. We also collected 54,738 debugging events spread over 192 users, 218 projects and 723 IDE sessions. In total, we recorded both at least one debug interval and one event for 108 users. The number of occurrences of the different event types as well as the types of breakpoints added and the types of breakpoint changes can be found in Table 6.1.

As the results above indicate, only 132 of the 458 users (28.8%) used the IDE debugger during the data collection period even though their IDE actually provides one, with no significant difference between Eclipse (28.9%) and IntelliJ (27.6%) users. Moreover, only 108 WatchDog 2.0 users (23.6%) have used the debugger and at least one of its features.

[4] http://stats.oecd.org/index.aspx?DataSetCode=ANHRS

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

Table 6.1: Frequency tables of received events as well as the breakpoint types and options seen in them.

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

The results shown in Table 6.1 indicate that line breakpoints are by far the most used breakpoint type. The other, likely more advanced, types account for less than 7% of all breakpoints set during the collection period. Furthermore, line breakpoints are used by most developers using the debugging infrastructure, while the other types of breakpoints are used by only $7.6 - 20.5\%$ of these developers.

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

Looking at the breakpoint change type frequencies in Table 6.1, we see that almost all of these changes are related to the enablement of the breakpoints. The other change types account for only 10.9% of all breakpoint changes. Furthermore, the number of users that generated these events range from 1 (0.8%) to 12 (9.1%). Moreover, the events related to specifying a hit count on the breakpoint have not been recorded at all during the collection period.

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

Table 6.1 shows that most of the recorded events are related to the evolution of breakpoints, hitting them during debugging and stepping through the source code. The more advanced debugging features like defining watches and modifying variable values have been used much less. Furthermore, the same holds for the number of users generating these events: the majority of users have added and/or removed breakpoints and stepped through the code, while only $2.3 - 15.2\%$ modified variable values, evaluated expressions and/or defined watches.

Setting breakpoints and stepping through the code is done most and by most developers using the debugger, other debugging features are used much less and by much less developers.

6.2.2 RQ2.1: How much of their active IDE time do developers spend on debugging (compared to other activities)?

For **RQ2.1**, we first computed the total duration of all intervals of a particular type and based it on the total duration of 'IDE open' intervals (18,156.9 hours, 100%) in the collection period. We recorded 25.2 hours of running unit tests (0.14%), 721.5 hours of debugging intervals (3.97%), 2,568.8 hours of reading (14.15%), and 1,228.6 hours of typing (6.77%). More generic interval types include e.g. 'WatchDog open' (0.12%) and 'IDE active' (28.92%). Next, we analysed the durations and percentages on a per user basis. For the users with at least one debug interval, the descriptive statistics of the resulting durations and percentages are shown in Table 6.2. This table also shows the corresponding values for reading, typing and JUnit intervals.

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

Table 6.2: Descriptive usage statistics for important interval types.

Looking at the results shown in Table 6.2 and the fact that the total recorded active IDE time was 5250.7 hours, we see that, in total, debugging consumes 13.7% of the active in-IDE development time, while reading or writing production or test code and running tests take 48.6%, 23.4% and 0.5%, respectively.

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

6.2.3 RQ2.2: What is the (average) frequency and length of the debugging sessions performed by developers?

For **RQ2.2** we first analysed the number of debug intervals per user for the 132 developers that have used the debugger during the collection period. The resulting numbers range from a single debug interval to 598 debugging sessions, with an average of 23.8 and a median of 4 debug intervals per user. Next, we analysed the durations of the $3,142$ debug intervals and found values ranging from 3 milliseconds to 90.8 hours, with an average and median duration of 13.8 minutes and 42.3 seconds, respectively.

Based on these results, we find that about half of the users using the IDE-provided debugging infrastructure have launched the debugger four times or less during the two months of data collection, while 21% of the developers launched their debugger more than 20 times, making them responsible for over 80% of the debugging sessions.

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

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

Most debugging sessions do not take much time, only 12% of them take more than ten minutes.

6.2.4 RQ2.3: At what times do developers launch the IDE debugger?

In order to find an answer to **RQ2.3**, we assessed the intervals that occur right before a debugging session is started. The resulting frequencies and their percentages of all intervals occurring before any debug interval are: 119 (1.24%) for other debug intervals, 46 (0.48%) for running unit tests, $4,991$ (51.94%) for reading and $1,802$ (18.75%) for typing intervals.

Therefore, we find that about 70% of the debugging sessions are started after reading or writing code. Furthermore, only 0.5% of them are started after a failing or passing test run.

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

6.2.5 RQ2.4: Do long files require more debugging?

For **RQ2.4**, we researched whether there is a correlation between the file size of a class (in SLOC [28]), and the number of times it is used for debugging. At $\rho = -0.75$, we find a strong negative correlation. We also investigated the relation between the file sizes and the duration of the debug intervals in which they are opened and found no correlation ($\rho = 0.19$).

> Classes are opened more during debugging as they get smaller.

6.2.6 RQ2.5: Is a small set of classes responsible for most debugging effort?

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

For 1., we found that on average, only 4.83% (median: 1.66%) of all project classes were ever debugged. The value ranges from 0.22% to 100%, where low values are found in projects with very few classes while the 100% cases stem from projects with only one or two classes. For 2., the results range from 0.81% to 100% with an average of 14.47% (median: 4.55%). Both results seem to indicate that debugging is focused on a relatively small set of classes in the project.

In addition, we find that in 75% of the debugging intervals at most 5% of the project's classes are opened. Furthermore, in 75% of these intervals, less than 15.8% of the classes that are ever debugged in the project are opened.

> In most cases less than 5% of the project's classes are debugged in a debug interval.

6.2.7 RQ2.6: Do developers who test a lot have to debug less?

In **RQ2.6**, we first investigated the relation between the total duration of running unit tests and the debug intervals per user. We performed a Spearman rank-order correlation test using the 144 developers having at least one debug interval or one unit testing interval, resulting in no correlation ($\rho = 0.11$). Then, we only considered the 25 developers with at least one debug interval and one unit testing interval. At $\rho = 0.58$, we find a moderate correlation between the two durations.

> Developers who spend a lot of time running tests are likely to debug a lot as well.

Next, we studied the relation between the amount of time the user spends inside test classes and the debugging time, again by performing a Spearman rank-order correlation

test. For the 248 developers with at least one debug interval or one opened test class, we find no correlation at $\rho = -0.08$. Furthermore, we find no correlation ($\rho = 0.23$) when focussing on the 84 users with both at least one debug interval and one opened test class.

> Developers who spend a lot of time within test classes are not likely to debug less or more.

6.2.8 RQ2.7: Do experienced developers have to debug less?

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

> More experienced developers are likely to spend a bit more time debugging.

6.2.9 RQ3.1: Which sequences of events are commonly found within and between debugging sessions?

For **RQ3.1** we first created a frequency table of the sequences of events that occurred during a debugging session. Next, we did the same for the sequences in which all consecutive events of the same type are collapsed into a single event in order to reduce the number of sequences that only occur once. Finally, the sequences were compressed even further by collapsing all different kinds of stepping events into a single one. Parts of these three frequency tables are shown in Table 6.4. For a description of the acronyms used, see Table 6.3.

Acronym	Description
BA	Breakpoint added
BR	Breakpoint removed
CC	Condition changed
CE	Condition enabled
DS	Breakpoint disabled
EN	Breakpoint enabled
RC	Resumed by client
SB	Suspended by breakpoint
SO	Stepped over
STEP	Stepped over, into or out

Table 6.3: Descriptions of used acronyms for events.

Complete	**Frequency**
No events	1995
SB	89
SB→RC	48
SB→SB	25
RC→SB	18
864 other sequences	967
	3142

Compressed	**Frequency**
No events	1995
SB	125
SB→RC	56
RC→SB	24
SB→SO	20
722 other sequences	922
	3142

More compressed	**Frequency**
No events	1995
SB	125
SB→RC	56
SB→STEP	35
STEP→SB→STEP	28
662 other sequences	903
	3142

Table 6.4: Complete and compressed event sequences within debug intervals.

In all cases shown in Table 6.4, the empty sequence occurs most, which means that no debugging features were used during the debug interval. Next, the sequence consisting of a single 'suspended by breakpoint' event had the highest frequency, followed by the sequence of 'suspended by breakpoint' followed by 'resumed by client'. In the tables of both compressed sequences, these two sequences were followed by the sequences consisting of a repetition of the sequence 'suspended by breakpoint' followed by a stepping event.

Next, we did a similar analysis for the events that occur between two debug intervals, in which events occurring after a single interval are ignored. We also compressed these sequences once in the same way as described above. The most frequent sequences and their frequencies are presented in Table 6.5. The descriptions of the acronyms can again be found in Table 6.3.

In both cases shown in Table 6.5, the most frequent sequence was again the empty sequence, followed by the sequence consisted of the addition of at least one breakpoint. Following these sequences are the sequences consisting of several combinations of breakpoint additions and removals.

No debugging features are used within and between most debugging sessions.

Programs are mostly suspended and stepped through during debugging and breakpoints are mainly added in between debugging sessions.

Complete	Frequency	Compressed	Frequency
No events	2548	No events	2548
BA	102	BA	134
BA→BA	28	SB	27
SB	20	BR	25
BR	18	BA→BR	18
333 other sequences	426	285 other sequences	390
	3142		**3142**

Table 6.5: Complete and compressed event sequences between debug intervals.

6.2.10 RQ3.2: How do breakpoints evolve over time?

For **RQ3.2** we first analysed the sequences of breakpoint events in a similar way as for **RQ3.1**, but per IDE session instead of per debugging session. Parts of the resulting frequency tables for the cases of complete and compressed sequences can be found in Table 6.6, which uses the acronyms from Table 6.3.

Complete	Frequency	Compressed	Frequency
No events	344	No events	344
BA	24	BA	38
BA→BR	16	BA→BR	31
BR	9	BA→BR→BA	17
BA→BA	9	BR→BA	17
174 other sequences	321	71 other sequences	276
	723		**723**

Table 6.6: Complete and compressed breakpoint event sequences per IDE session.

Table 6.6 shows that the most frequent sequence after the empty sequence is in both cases the addition of at least one breakpoint, followed by the sequence(s) consisting of at least one breakpoint addition followed by at least one breakpoint removal.

To see how breakpoints are changed over time, we also looked at the sequences of breakpoint changes during the IDE sessions in the same way. The resulting frequency tables are (partially) presented in Table 6.7, using the acronyms from Table 6.3.

Based on the results in Table 6.7, we find that the non-empty sequences found most are the ones in which one or more breakpoints are disabled, followed by the sequences in which a condition is added to one or more breakpoints.

Most breakpoints that are changed over time are disabled or become conditional.

Complete	Frequency	Compressed	Frequency
No events	578	No events	578
DS	5	DS	14
DS→DS	4	CE	5
CE	3	DS→EN→DS→...	2
DS→DS→DS	2	CE→CC→EN→...	1
16 other sequences	131	9 other sequences	123
	723		**723**

Table 6.7: Complete and compressed sequences of breakpoint changes per IDE session.

6.2.11 RQ3.3: How often does it occur that a developer steps over the point of interest and has to start all over again?

During our research into debugging, we sometimes heard anecdotal reports of frustrated developers stepping over the point of interest while debugging. To this end, we seeked objective data to support how severe of a problem it really is by looking for possible cases of stepping over the point of interest for **RQ3.3**, which means that the developer steps one time too many and has to start debugging all over again. For this we created a set of debug intervals that satisfy the following two conditions: 1. the last event occurring within the debug interval is a stepping event; and 2. the interval is followed by another debug interval in the same IDE session. Then we created several subsets of this debug intervals by imposing a maximum time period between two consecutive debug intervals. Figure 6.1 shows the possible cases of stepping over the point of interest for the subsets with a maximum time period between one second and 15 minutes.

The trend line in Figure 6.1 shows that the amount of new possible cases of stepping over the point of interest starts to decrease significantly after about four minutes. At this point, about 150 possible cases can be identified, which corresponds to 4.8% of the debugging intervals.

Developers might step over the point of interest and have to start over again in less than about 5% of the debugging sessions.

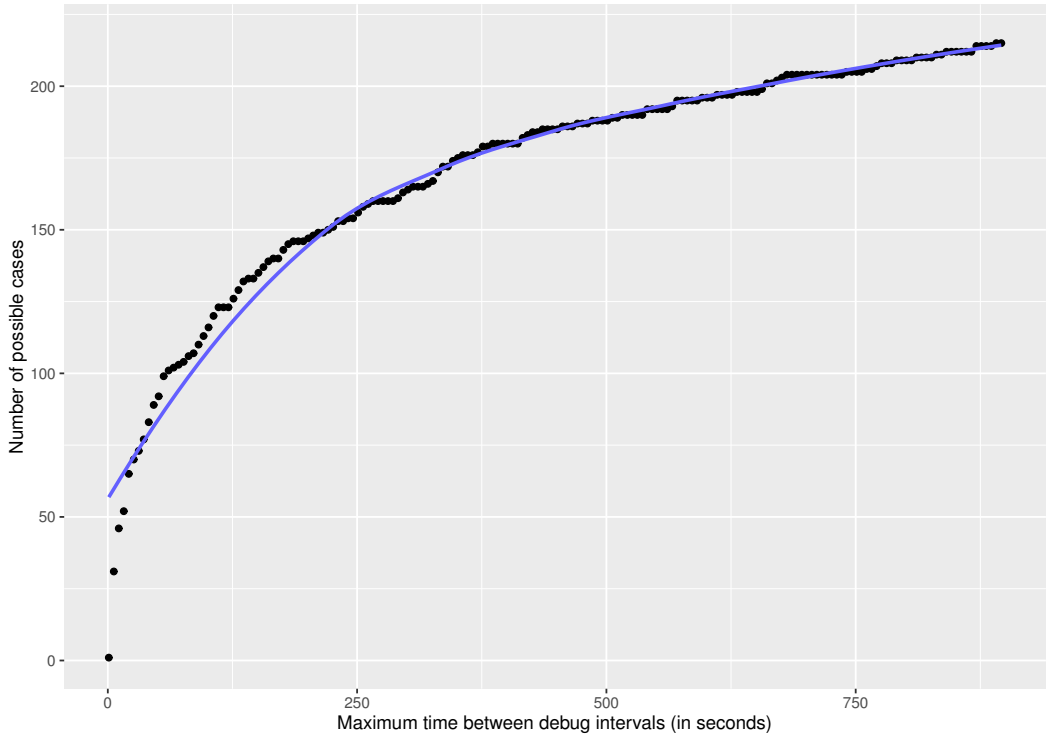

Figure 6.1: Possible cases of stepping over the point of interest per maximum time period between consecutive debug intervals.

Chapter 7

Discussion

This chapter first discusses the results obtained in Chapters 4 and 6 on a more abstract level and by comparing the results of both datasets. Next, we describe the limitations and possible threats affecting the validity of our study.

7.1 Interpretation of results

Usage of the IDE Debugger. In the results described in Chapter 4, we found that the vast majority of survey respondents claims to be using the IDE-provided debugging infrastructure and the integrated debugging features. However, in the results for WatchDog 2.0 we found that most developers are actually not using their IDE debugger and its features. This contradiction points to a difference between the perception and reality of debugging behavior, similar to what Beller et al. observed for testing [5]. However, although unlikely, it might be that most of the observed developers did not have to fix a bug during the observation period.

Furthermore, the low observed use of the IDE-provided debugging infrastructure is a result in itself. In the survey, many respondents claimed to be using print statements and log files. In addition, many respondents that are not using the IDE debugger find techniques like these more effective or efficient. By analysing the responses to the statement that "the best invention in debugging still was printf debugging", we also found that some like using print statements over IDE debuggers as they are fast and easy, others like using tests to avoid debugging better and some believe that debuggers are jittery or not suitable in concurrent environments or have too much overhead. Therefore, these results might explain the low observed debugger usage. However, other respondents find that using debuggers is required as printf debugging is not enough (for harder bugs) or they find them simply better. These differences of opinion point to a division in the developer community that needs to be explored further in future research.

Usage of Debugging Features. The survey results also indicated that most developers use line breakpoints, but do not use more advanced breakpoint types like class prepare breakpoints. The same conclusion can also be drawn from the data obtained with WatchDog 2.0. This raises the following question: do developers find the unused breakpoint types

useless or are they simple unknown? Figure 4.1 shows that most developers actually claim to know most of these other breakpoint types. So, are these types actually useless or do developers that know them not know how to apply them in practice?

Regarding the usage of breakpoint options, we found that most developers are familiar with specifying conditions, but not with setting the hit count or suspend policy. A similar result was extracted from WatchDog 2.0's data. However, the number of developers using them was much lower than the survey indicated, as shown in Figure 4.2. This gap might be explained by the fact that the survey was targeted at debugging specifically, which might have attracted more experienced users than in the general WatchDog 2.0 population.

A similar result is visible in both the survey and WatchDog 2.0 data about the usage of other debugging features like stepping through the code. In both cases we found that these features get used less at they get more advanced. However, the observed numbers on the usage of these features are much lower than the claimed usage visualized in Figure 4.3. For example, while 60% of the survey respondents indicated to define watches during debugging, only 15.2% of the WatchDog 2.0 users that use the debugger have defined a watch.

Usage of Tests for Debugging. In the survey, most respondents think (unit) testing is an integral part of the debugging process, especially for reproducing bugs at the beginning of the process. Moreover, one respondent even thinks that "the best invention is to eliminate the need for debugging by maintaining a test suite". However, this usage of tests is not visible in the data obtained with WatchDog 2.0 as shown by the results for **RQ2.3**. Regarding the results of **RQ2.6** which relates the time spend on testing with debugging, we found that developers spending more time on running tests in their IDE are likely to spend more time on debugging. This might indicate that these developers often execute tests during the debugging process. However, this is again not supported by the results for **RQ2.3** as well as by the other results for **RQ2.6**. Therefore, developers do not seem to actually use tests during the debugging process. This raises the following question: why are they not using tests for debugging or do they use and/or define these tests outside of the IDE?

Effects of Experience on Debugging. Based on the survey responses, we found that experience in software development has limited to no impact on whether or not the developer uses the IDE debugger and its features as well as (unit) tests. However, one survey respondent indicates that he or she has "the feeling that as I get more experienced I use the debugger less and less". Nevertheless, the results for **RQ2.7** obtained with WatchDog 2.0 also report a limited impact of programming experience on the time spend on debugging. Therefore, there seems to be no significant relation between experience and debugging behavior within the IDE, which, combined with previous results, might mean that experienced developers do not absolutely have more knowledge on IDE debugging than less experienced ones. This again might point to a lack of education on debugging amongst many software developers ranging from beginners and CS students to experts.

Dependencies between Debugging Features. In the analysis of the survey results, we found that developers claiming to use line breakpoints likely also step through the code. The same relation is visible in the data obtained with WatchDog 2.0. In fact, these two events are part of common debugging behavior as indicated by e.g. Tables 6.1 and 6.4.

Results of Sub Research Questions. For **RQ2.1** we found that developers, in total, spend less than 4% of the time their IDE is open on debugging and less than 14% of the

time they are actively working in the IDE. If you would consider all tracked users as being part of a single project, then the result is quite a bit lower than the $30-90\%$ for testing and debugging as reported by Beizer [3]. So, is debugging not such a "dirty little secret" [22] after all or is much of the debugging effort performed outside of the IDE?

We also found in **RQ2.4** that classes are likely to be visited during debugging more as they get smaller, which is quite surprising as long classes are often considered as a 'code smell' [13]. However, we found no correlation between the length of a file and the amount of required debugging time.

In the results for **RQ2.5** we found that only a small set of classes is often responsible for all recorded debugging effort. This result might be used as a starting point for developing a new or improved automated debugging technique by e.g. keeping track of this set of classes and only show the results of slicing or spectra-based fault localization for the statements in this set.

Finally, we found that in less than about 5% of the debugging sessions the developer might have stepped over the point of interest and had to start debugging all over again. This indicates that there is a limited, but existent gap in the debugging process that might be filled with so-called *back-in-time debuggers* (e.g. [40]) that allow developers to step back in the program execution in order to arrive at the point of interest.

7.2 Threats to validity

In this section we discuss both the limitations as well as the construct, internal and external validity of our study.

Limitations of this study include the fact that over 80% of the survey respondents seems to be from the Java community. Therefore, little survey data is available about communities focussing on other programming languages. The same holds for the analysis of the WatchDog 2.0 data, as our tool is currently only available for Eclipse and IntelliJ. Therefore, data generated by other (Java) IDEs is not included in our analysis.

Furthermore, the group of survey respondents is different from the group of WatchDog 2.0 users we tracked. This means that the results obtained by both methods stem from different populations, which makes a comparison of these results harder.

Construct validity deals with the threats originating from the way in which we collect our data. As the WatchDog 2.0 implementation is build on top of the original WatchDog version, possible errors in the original tool might still exist. Furthermore, the implementation of the new functionality is prone to human errors.

To minimize these risks, the automated test suite of WatchDog was extended to also test for the correctness of the new functionality. Furthermore, we use this test suite for regression testing to make sure existing functionality is not broken throughout the process. In addition, extensive manual testing has been performed to verify whether or not the intervals and events are properly generated, stored and transferred. Finally, we performed code reviews before changes were actually integrated.

Internal validity is concerned with the threats that are inherent to the study described in this report. The first threat is that we might not have found common debugging issues on StackOverflow because StackOverflow's question guidelines[1][2] forbid users to ask general, open-ended questions. Furthermore, although we have run the analyses with different numbers of topics, the arbitrarily chosen number of topics might still have led to imprecise results.

The first threat regarding the survey analysis is the risk that one person under- and/or overuses some of the tags in the open card sorting process. To mitigate this risk, we first performed the card sorting with two persons in close collaboration. Afterwards, we sampled some of the responses and had another person sort the cards, independently of the existing tags. Then, we compared the results and checked the inter-rater agreement. We found that the differences between the results were mainly caused by having different interpretations of some tags.

Furthermore, the mentioned set of different answers to the open survey questions based on analysing the tags resulting from the open card sorting cannot be taken as conclusive. This is because respondents might (not) have thought about and mentioned these key words at random. Therefore, these answer types only serve as an indication of their opinions.

Finally, as WatchDog 2.0 is not (yet) able to determine which classes are being debugged, we approximated this for **RQ2.4** and **RQ2.5** by looking at the classes that are opened for reading or typing during a debug interval. However, as it seems reasonable to assume that a possible bias is constant across all debug intervals, the ordering of the resulting percentages is not affected by such a bias. A similar argument holds for approximating the number of classes in a project by counting the unique classes ever seen in an interval for that project.

External validity deals with the extent to which the results of this study are generalizable. During a data collection period of more than two months we collected $1,155,189$ intervals with a total duration of over ten developer years, spread over 458 users. However, the results do likely still not hold for all individual developers and organizations. Furthermore, the collected data is only generated by Eclipse and IntelliJ, meaning that the results could not hold for other Java IDEs. Moreover, they could be very different for users outside of the Java community or when including debugging behavior outside of the IDE.

[1] http://stackoverflow.com/help/dont-ask
[2] http://stackoverflow.com/help/on-topic

Chapter 8

Conclusions and Future Work

As debugging is still necessary and automated techniques are not there yet (see e.g. [29]) and are also not widely used [35], techniques like symbolic debugging in the IDE are still required. However, we are not aware of the existence of any study on the debugging behavior of software developers in IDEs. To this end, we studied the developers' perception on (IDE) debugging by conducting an online survey after having searched for common issues with debugging on StackOverflow. In addition, we looked for (the absence of) common debugging behavior by instrumenting Eclipse and IntelliJ to collect behavioral data using WatchDog 2.0.

Based on the resulting datasets, we found several similarities as well as differences and contradictions between the developers' perception on their debugging habits and the actual IDE debugging behavior. For instance, the vast majority of the survey respondents claims to be using the IDE-provided infrastructure, but we found that the vast majority of developers we collected data from does not actually use this infrastructure. Furthermore, we found several other, some even surprising, results based on using just one of the two datasets. For example, we found no correlation between the time spend in test classes and the required debugging time. In addition, we found that programming experience has limited to no impact on the usage of IDE-provided debugging features. In general, our study has the following implications:

Software Developers should be aware that they tend to overestimate their usage of the IDE-provided debugging infrastructure. Furthermore, they should know that more debugging features and techniques exist and can be applied in practice than the ones they are using.

IDE creators should know that the debugging features they provide are likely not used by most developers. Moreover, they should be aware that some features are used far more frequent than others. This knowledge could be used to, e.g., give them a more prominent place in the IDE. In addition, they should know that developing or integrating back-in-time debuggers might fill a gap in the debugging process.

Educators/Reseachers should be aware that developers' perceptions can be quite different from their actual behavior. Furthermore, they should know that some common beliefs

on debugging and testing do not seem to hold in practice. Finally, they should be aware of the apparent lack of knowledge on debugging among software developers, which probably requires a more prominent place for debugging in curricula.

Based on the respondents' answers to the open survey question as well as the results of the survey and the WatchDog 2.0 data in general, we formulated several preliminary conclusions and hypotheses. To test these hypotheses, we plan to interview several developers and IDE creators. Furthermore, to increase the generalizability of this study, we plan to collect WatchDog 2.0 data over a longer period of time. Next, to overcome the limitation of only collecting data on Eclipse and IntelliJ and therefore mainly Java developers, we would like to support even more IDEs, especially non-Java ones. Finally, to be able to better compare the perception on debugging of developers and their debugging behavior, we plan to link the survey responses to the WatchDog 2.0 data to have data originating from the same population. We already set up the infrastructure to support this. However, at the time of writing, too few responses were available to draw significant results.

In conclusion, the hypotheses and results we obtained in this study need to be tested and researched more in future work to increase their credibility and generalizability. However, the surprising results of this study show that we might need to review our commonly accepted beliefs on debugging and open up new possibilities for research.

Bibliography

[1] David Abramson, Clement Chu, Donny Kurniawan, and Aaron Searle. Relative debugging in an integrated development environment. *Software: Practice and Experience*, 39(14):1157–1183, 2009.

[2] Anton Babenko, Leonardo Mariani, and Fabrizio Pastore. Ava: Automated interpretation of dynamically detected anomalies. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 237–248, New York, NY, USA, 2009. ACM.

[3] Boris Beizer. Software testing techniques. *New York, ISBN: 0-442-20672-0*, 1990.

[4] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. When, how, and why developers (do not) test in their ides. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 179–190, New York, NY, USA, 2015. ACM.

[5] Moritz Beller, Georgios Gousios, and Andy Zaidman. How (much) do developers test? In *Proceedings of the 37th International Conference on Software Engineering (ICSE), NIER Track*, pages 559–562. IEEE, 2015.

[6] Moritz Beller, Igor Levaja, Annibale Panichella, Georgios Gousios, and Andy Zaidman. How to catch em all: Watchdog, a family of ide plug-ins to assess testing. In *ICSE 2016, Austin, USA, 14-22 May 2016; Authors version*. ACM/IEEE, 2016.

[7] David M Blei. Probabilistic topic models. *Communications of the ACM*, 55(4):77–84, 2012.

[8] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *the Journal of machine Learning research*, 3:993–1022, 2003.

[9] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. Angelic debugging. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 121–130. IEEE, 2011.

[10] Marcel Das, Peter Ester, and Lars Kaczmirek. *Social and behavioral research and the internet: Advances in applied methods and research strategies*. Routledge, 2010.

[11] Frank Eichinger, Klaus Krogmann, Roland Klug, and Klemens Böhm. Software-defect localisation by mining dataflow-enabled call graphs. In *Machine Learning and Knowledge Discovery in Databases*, pages 425–441. Springer, 2010.

[12] Frank Eichinger, Victor Pankratius, Philipp WL Große, and Klemens Böhm. Localizing defects in multithreaded programs by mining dynamic call graphs. In *Testing–Practice and Research Techniques*, pages 56–71. Springer, 2010.

[13] Martin Fowler, Kent Beck, J Brant, William Opdyke, and Don Roberts. Refactoring: Improving the design of existing programs, 1999.

[14] James Gleick. Little bug, big bang. `http://www.nytimes.com/1996/12/01/magazine/little-bug-big-bang.html`, 1996. [Online; accessed 30 June 2016].

[15] Lars Heinemann, Benjamin Hummel, and Daniela Steidl. Teamscale: Software quality control in real-time. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 592–595. ACM, 2014.

[16] Will G Hopkins. A new view of statistics. `http://newstatsi.org`, 1997. [Online; accessed 16 March 2015].

[17] Andrew Hunt and David Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Professional, 2000.

[18] InfoEther Inc. Pmd - finding copied and pasted code. `http://pmd.sourceforge.net/pmd-4.3.0/cpd.html`. [Online; accessed 3 May 2016].

[19] A Lehman, N ORourke, L Hatcher, and EJ Stepanski. Jmp for basic univariate and multivariate statistics: A step-by-step guide. sas institute. *Inc., Cary, NC*, 2005.

[20] Yan Lei, Xiaoguang Mao, Ziying Dai, and Chengsong Wang. Effective statistical fault localization using program slices. In *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual*, pages 1–10, July 2012.

[21] Jie Li, Changhai Nie, and Yu Lei. Improved delta debugging based on combinatorial testing. In *Quality Software (QSIC), 2012 12th International Conference on*, pages 102–105, Aug 2012.

[22] Henry Lieberman. The debugging scandal and what to do about it (introduction to the special section). *Commun. ACM*, 40(4):26–29, 1997.

[23] Lucia Lucia, David Lo, Lingxiao Jiang, Ferdian Thung, and Aditya Budi. Extended comprehensive study of association measures for fault localization. *Journal of Software: Evolution and Process*, 26(2):172–219, 2014.

[24] Muhammad Zubair Malik, Junaid Haroon Siddiqi, and Sarfraz Khurshid. Constraint-based program debugging using data structure repair. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 190–199. IEEE, 2011.

[25] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[26] Andrew Kachites McCallum. Mallet: A machine learning for language toolkit. http://mallet.cs.umass.edu, 2002.

[27] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM Trans. Softw. Eng. Methodol.*, 20(3):11:1–11:32, August 2011.

[28] Vu Nguyen, Sophia Deeds-Rubin, Thomas Tan, and Barry Boehm. A sloc counting standard. In *COCOMO II Forum*, volume 2007, 2007.

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

[30] Saeed Parsa and Somaye Arabi Naree. A new semantic kernel function for online anomaly detection of software. *ETRI Journal*, 34(2):288–291, 2012.

[31] Saeed Parsa, Mojtaba Vahidi-Asl, Somaye Arabi, and Behrouz Minaei-Bidgoli. Software fault localization using elastic net: A new statistical approach. In *Advances in Software Engineering*, pages 127–134. Springer, 2009.

[32] Saeed Parsa, Mojtaba Vahidi-Asl, and Maryam Asadi-Aghbolaghi. Hierarchy-debug: a scalable statistical technique for fault localization. *Software Quality Journal*, 22(3):427–466, 2014.

[33] Saeed Parsa, Farzaneh Zareie, and Mojtaba Vahidi-Asl. Fuzzy clustering the backward dynamic slices of programs to identify the origins of failure. In *Experimental Algorithms*, pages 352–363. Springer, 2011.

[34] Michael Perscheid and Robert Hirschfeld. Follow the path: Debugging tools for test-driven fault navigation. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 446–449. IEEE, 2014.

[35] Michael Perscheid, Benjamin Siegmund, Marcel Taeumel, and Robert Hirschfeld. Studying the advancement in debugging practice of professional software developers. *Software Quality Journal*, pages 1–28, 2016.

[36] Fabio Petrillo, Zphyrin Soh, Foutse Khomh, Marcelo Pimenta, Carla Freitas, and Yann-Gal Guhneuc. Understanding interactive debugging with swarm debug infrastructure. In *Proceedings of the 24th International Conference on Program Comprehension*, pages 1–4. ACM, 2016.

[37] David Piorkowski, Scott Fleming, Christopher Scaffidi, Christopher Bogart, Margaret Burnett, Bonnie John, Rachel Bellamy, and Calvin Swart. Reactive information foraging: An empirical investigation of theory-based recommender systems for programmers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pages 1471–1480, New York, NY, USA, 2012. ACM.

[38] David Piorkowski, Scott D. Fleming, Christopher Scaffidi, Margaret Burnett, Irwin Kwan, Austin Z. Henley, Jamie Macbeth, Charles Hill, and Amber Horvath. To fix or to learn? how production bias affects developers' information foraging during debugging. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, ICSME '15, pages 11–20, Washington, DC, USA, 2015. IEEE Computer Society.

[39] David J Piorkowski, Scott D Fleming, Irwin Kwan, Margaret M Burnett, Christopher Scaffidi, Rachel KE Bellamy, and Joshua Jordahl. The whats and hows of programmers' foraging diets. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3063–3072. ACM, 2013.

[40] Guillaume Pothier and Éric Tanter. Back to the future: Omniscient debugging. *IEEE software*, 26(6):78–85, 2009.

[41] Allen B Riddell. How to read 22,198 journal articles: Studying the history of german studies with topic models. *Distant Readings: Topologies of German Culture in the Long Nineteenth Century. Camden House*, pages 91–114, 2014.

[42] Jeremias Rößler, Gordon Fraser, Andreas Zeller, and Alessandro Orso. Isolating failure causes through test case generation. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 309–319, New York, NY, USA, 2012. ACM.

[43] Eric Schmitt. U.s. details flaw in patriot missile. `http://www.nytimes.com/1991/06/06/world/us-details-flaw-in-patriot-missile.html`, 1991. [Online; accessed 30 June 2016].

[44] Benjamin Siegmund, Michael Perscheid, Marcel Taeumel, and Robert Hirschfeld. Studying the advancement in debugging practice of professional software developers. In *Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on*, pages 269–274. IEEE, 2014.

[45] Donna Spencer. *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.

[46] Richard Stallman, Roland Pesch, Stan Shebs, et al. *Debugging with GDB*. Free Software Foundation, 10 edition, 2011.

[47] Roykrong Sukkerd, Ivan Beschastnikh, Jochen Wuttke, Sai Zhang, and Yuriy Brun. Understanding regression failures through test-passing and test-failing code changes. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1177–1180, Piscataway, NJ, USA, 2013. IEEE Press.

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

[49] TestRoots. Testroots - watchdog eclipse plugin. `https://testroots.org/testroots_watchdog.html`. [Online; accessed: 4 July 2016].

[50] Jane Wakefield. Heartbleed bug: What you need to know. `http://www.bbc.com/news/technology-26969629`, 2014. [Online; accessed 30 June 2016].

[51] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 61–72, New York, NY, USA, 2010. ACM.

[52] Kai Yu, Mengxiang Lin, Jin Chen, and Xiangyu Zhang. Practical isolation of failure-inducing changes for debugging regression faults. In *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, pages 20–29. IEEE, 2012.

[53] Kai Yu, Mengxiang Lin, Jin Chen, and Xiangyu Zhang. Towards automated debugging in software evolution: Evaluating delta debugging on real regression bugs from the developers perspectives. *Journal of Systems and Software*, 85(10):2305 – 2317, 2012. Automated Software Evolution.

[54] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10. ACM, 2002.

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

[56] Andreas Zeller and Dorothea Lütkehaus. Ddda free graphical front-end for unix debuggers. *ACM Sigplan Notices*, 31(1):22–27, 1996.

[57] Cheng Zhang, Juyuan Yang, Dacong Yan, Shengqian Yang, and Yuting Chen. Automated breakpoint generation for debugging. *Journal of Software*, 8(3), 2013.

[58] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Locating faulty code by multiple points slicing. *Software: Practice and Experience*, 37(9):935–961, 2007.

[59] Yiji Zhang and Raul Santelices. Prioritized static slicing for effective fault localization in the absence of runtime information. Technical report, Technical Report TR 2013-06, CSE, U. of Notre Dame, 2013.

Appendix A

Other Topic Modelling Results

This chapter contains all results of the topic modelling process with MALLET that are mentioned but not visually presented in Chapter 3. Presenting all results in Chapter 3 would overflow the chapter with tables due to the space required for all tables containing the topic modelling results. Therefore, these other results are presented here. Each of the following sections shows the results of a particular setting, which is a combination of the StackOverflow posts selected as input and the number of topics selected as output.

A.1 Topic modelling with all posts - 50 topics

The results visualised in this section are the results when using all posts selected by the process described in Section 3.1 as input and setting the number of topics in the output to 50. As the resulting table is too large to be readable when put on a single page, the results are split up into two parts. The first part can be found in Table A.1 and the second part in Table A.2.

A.2 Topic modelling with Java posts - 10 topics

This section presents the results of the topic modelling process with all posts in the Java category as input and 10 topics in the output. The results are shown in Table A.3.

A.3 Topic modelling with Java posts - 30 topics

Similar to the previous section, this section shows the results when using the Java posts as input, but now with 30 topics in the output. As the resulting table is again too large for a single page, Table A.4 shows the first part and Table A.5 the second.

A.4 Topic modelling with general posts - 10 topics

This section presents the results of the topic modelling process with all posts in the general category as input and 10 topics in the output. The results are shown in Table A.6.

Topic #	Weight	Key words	Possible descriptive label
0	0,22218	code error debug line i'm problem debugger exception debugging run program works call method application fine message function i've doesn't	General debugging
1	0,00551	e/androidruntime i/debug error/androidruntime d/dalvikvm freed w/system.err method paused free info/debug i/system.out pid thread i/dalvikvm activity total ffffffff exception android debugger	Android debugging
2	0,0073	error android build file project compile javac gradle ant failed class jar task package files source dependencies find target echo	Android building
3	0,01687	int float height width image void game return true color vector public position private false player point draw screen size	C/C++ game development
4	0,01212	nsstring alloc view nil return app error data property init cell code release ios method tableview controller import nsarray object	iOS development
5	0,0126	row data string column table true rows datatable grid int col code columns dataset false return null cell select gridview	.NET data management
6	0,01656	php echo line function array error file xdebug version zend warning script return true false extension load module enabled isset	PHP debugging
7	0,03047	service request response http client wcf server post error url json data web string message code header call method headers	HTTP errors
8	0,00185	call actionpack app[web block vendor/bundle/ruby activesupport rack lib/active_support/callbacks.rb railties gems/actionpack process_action activerecord lib/rails/rack/logger.rb instrument end lib/ruby run render lib/action_dispatch/routing/route_set.rb gems/rack	Ruby building
9	0,00889	info error maven jar debug compile project build plugin pom errors failed snapshot repository true test flash found mvn flex	Maven errors
10	0,0098	map google location api token oauth twitter maps code key longitude status lat latitude null marker provider mapview double analytics	Google maps/Twitter APIs
11	0,01114	java eclipse jar report source bundle native env class debug plugin scala jni ndk message ember play val error cocos	Eclipse Scala plugin
12	0,02251	list node item xml null product items element parent child return title tree document category data nodes root search index	XML tree navigation
13	0,0321	var function return data true false ajax jquery javascript json code undefined object success type html url error width div	Web development (JavaScript)
14	0,00802	info debug source main tomcat thread java server aug error nov jar starting start sep jul oct true failed root	Tomcat log
15	0,00705	key certificate ssl symfony hash sha keystore signature false cipher cbc_sha true byte length client openssl password path error listener	SSL authentication
16	0,04539	thread time memory process seconds threads running task start run application queue main timer worker long data lock loop bytes	Timing issues
17	0,0267	file image files filename upload path data read directory byte string code size images open folder download stream write null	File IO
18	0,01325	error include const void int lnk class function unresolved symbol public external return char qdebug referenced bool studio template operator	C/C++ Qt errors
19	0,06388	page html url javascript facebook browser chrome code debug link content load script site works firefox jquery working problem button	Web development
20	0,0131	public void import override e/androidruntime private activity null int string return intent view class final button method extends textview protected	Android development
21	0,04091	public class object return string private set method type void null static list property entity instance int objects override var	C# development
22	0,08287	project file studio files visual debug build dll windows library folder application release error solution run version configuration path source	Visual Studio building
23	0,00757	error apache server loadmodule django file client files debug default options static set configuration directory include http url settings admin	Apache settings
24	0,00573	npm err http error node install node.js code command verbose express silly gyp system found path module.js info failed tgz	Node.js installation errors
25	0,0271	database query table sql null select data mysql insert values sqlite result connection set update create key error row statement	SQL issues

Table A.1: Topic modelling results with all posts and 50 topics (part 1).

26	0,01467	server client socket connection port data message send true connect byte received bytes tcp connected read buffer packet sending receive	Networking errors
27	0,02314	int char return include void struct cout null function size double code program unsigned const main define mov pointer endl	C/C++ development
28	0,24526	i'm code i've problem it's work debugging don't question time make data doesn't find issue edit working debug can't works	General debugging
29	0,04611	string array text number print return input function values line output code length result key data int type count character	Array/String issues
30	0,02107	public import string private void null static return catch final int class exception override throws ioexception e.printstacktrace boolean arraylist java	Java development
31	0,00968	undefined make checking reference error gcc include configure directory build file compiler mingw target library makefile compile version lib wall	Makefile/compiler issues
32	0,0109	date video audio stream play month player day null year int format time media calendar sound main ffmpeg playing file	Video player issues
33	0,02107	user password login username session debug request true return users filter authentication null email spring cookie false page string token	Web authentication
34	0,0095	debug email mail ssh smtp send message server host key password subject port true file type error sending body connection	SMTP errors
35	0,01194	gdb loaded fff symbols thread c:\windows\system frame dll corefoundation stack uikit ffff armv crash pdb file c:\windows\syswow find code exception	Windows development
36	0,02435	public model controller form return view array set data action string class mvc function true post required null field validation	C# MVC issues
37	0,02239	test command debug tests run script set file echo unit running git output shell files execute cmd path true line	Commandline issues
38	0,01529	file line import python def return print true class error django call traceback app c:\python webapp usr/local/lib/python recent usr/lib/python main	Python errors
39	0,01093	end dim set string function integer excel begin false code byval error true private vba debug.print object sheet range select	VB Excel development
40	0,07595	server application error web service windows running app local file debug iis machine remote connection asp.net access run debugging fine	.NET/IIS errors
41	0,00758	int null return intptr object boolean hwnd state uint callback bytes dll args msg bool lparam wparam window handle message	C/C++ Windows development
42	0,04308	app device android debug application ios phone xcode emulator run iphone sdk error eclipse devices version usb running debugging simulator	Debugging mobile apps
43	0,00931	gem end rails ruby require def true error lib/ruby/gems block assets false rake group bundle development app c:/ruby run production	Ruby building
44	0,0196	public string void exception var private static null return class int code catch sender object true task boolean false bool	C# development
45	0,00612	error warning file framework install referenced function found command clang failed setenv copying running python expected arch package gcc architecture	Python/Clang errors
46	0,00694	assembly version culture vec publickeytoken neutral opengl texture load shader boolean mono error null vertex reference type codebase context win	OpenGL issues
47	0,01692	log file logging debug logger level info message logs messages org.apache.log console output net perl configuration class error stdout appender	Logging issues
48	0,00717	com.opensymphony.xwork debug bean exception defaultactioninvocation.invoke(defaultactioninvocation.java spring error source method hibernate org.apache.struts transaction trace http jar null main session caused release	Java development
49	0,03023	event button text form control click void set property private window sender code item true label user textbox binding public	.NET UI development

Table A.2: Topic modelling results with all posts and 50 topics (part 2).

Topic #	Weight	Key words	Possible descriptive label
0	0,03253	info debug error log checking npm jar file java err version failed source maven spring install class build compile tomcat	Maven compilation errors
1	0,03251	public e/androidruntime void import string null private override int activity return class final android method i/debug static catch intent exception	Android development
2	0,03	file line import gem def error true python server loadmodule files return app false default debug require print options apache	Ruby/Python development
3	0,0503	debug server service request call client error end response http message app[web connection actionpack send user block wcf email mail	HTTP errors
4	0,02662	error file build files warning studio found project include loaded visual library directory c:\program dll make referenced reference version symbol	Visual Studio errors
5	0,03847	int return include void char const function null fff error code program thread struct main data size bytes unsigned double	C/C++ development
6	0,06758	function var data return array true page false query user table code null select url form error echo ajax jquery	Web development
7	0,03061	return var image view alloc width height nsstring float code nil int function error frame position false true color set	iOS development
8	0,14613	public class string return void object null method set private code list static exception var type int property end error	Java/C# development
9	0,35428	code i'm file problem error application i've debug app run work works time project it's set don't fine test debugging	General debugging

Table A.3: Topic modelling results with all Java posts and 10 topics.

A.5 Topic modelling with general posts - 30 topics

Similar to the previous section, this section shows the results when using the general posts as input, but now with 30 topics in the output. As the resulting table is again too large for a single page, Table A.7 shows the first part and Table A.8 the second.

A.6 Topic modelling with general posts - Breakpoint topic

The results of the topic modelling process when selecting all posts from the general category that belong to the subtopic about breakpoints are shown in Table A.9. As can be seen in the table, the number of topics in the output is 20.

A.7 Topic modelling with general posts - Java IDE topic

Similar to the previous section, the results shown in this section are based on a subtopic of the general category, but now the selected subtopic is ‘Java IDE’. The 20 resulting topics can be found in Table A.10.

A.8 Topic modelling with general posts - Watches topic

The results shown in Table A.11 are again obtained in a similar way as the previous two sections, but now for the subtopic that is about watches.

Topic #	Weight	Key words	Possible descriptive label
0	0,01129	checking error loadmodule file warning php install copying directory apache server framework files include version found running configuration options support	PHP errors
1	0,03625	var function true page data jquery ajax false javascript return html url json code options chrome script div type success	Web development
2	0,006	npm err setenv error http verbose install node athens buddha debug module.js express command node.js rebuild false message code info	Node.js errors
3	0,02268	debug info spring bean error exception tomcat method server application session hibernate jar thread log null true exec servlet configuration	Tomcat/hibernate errors
4	0,01746	xml document string file element error type node doc result attribute pdf true event match filter partial null xsd parse	XML parsing errors
5	0,02759	server client mail connection message email debug password port send error connect socket host true smtp address key username ssl	SMTP errors
6	0,02185	e/androidruntime public import void override activity null method string private android error/androidruntime class intent view int return final extends button	Android development
7	0,02911	exception task application catch null int var error thread boolean code async void return string stack await line object message	C# errors
8	0,03452	array return string list int case node null function break number false count input key true end data length code	Web development errors
9	0,01578	bytes data thread queue process read packet type set send start port len worker buffer qdebug received span pipe number	Network errors
10	0,13923	public class string object method return void static private null set type code list test var int property instance i'm	Java/C# development
11	0,01225	alloc nsstring view error nil return code data load init cell ios release delegate controller app property tableview method nsarray	iOS errors
12	0,01617	log file info logger debug logging import django line true error root false default level copied module class static python	Python logging errors
13	0,03723	table end database dim query sql select row string set data error date null echo function insert column values rows	SQL errors
14	0,0198	info error debug project jar build compile java android maven plugin file eclipse failed source run test class version package	Eclipse android plugin errors
15	0,0257	string public null catch private void final int static return byte file exception import ioexception thread data e.printstacktrace stream override	Java errors
16	0,04915	user model controller return public array view form set data null entity list var action function mvc true class email	.NET development
17	0,03521	void event public button form private item sender text code control false true page null click list class property set	C# development
18	0,01446	fff thread gdb armv memory kernel ios ffff unknown xcode corefoundation app crash symbols time file device warning stack version	Xcode errors
19	0,02065	image int float width height position return color void private bitmap false game left texture vec true frame background canvas	Game development
20	0,03181	int include char void return const function struct program unsigned error cout main code double define null endl file size	C/C++ errors
21	0,03255	request url response http page post facebook login server user error debug php token session header return api headers status	Facebook API authentication errors

Table A.4: Topic modelling results with all Java posts and 30 topics (part 1).

22	0,00956	i/debug d/dalvikvm freed paused android free i/system.out device info/debug error failed pid resource set total debugger i/dalvikvm waiting configuration debug/dalvikvm	Android debugging errors
23	0,01014	make undefined error reference gcc file include version build msvc target mingw directory configure compiler library makefile function compile found	Make compilation errors
24	0,04586	service web server error client wcf request string application iis asp.net response exception public code remote web.config net http message	WCF/IIS errors
25	0,03597	error project file studio visual build files dll loaded c:\program windows assembly debug library win version lib microsoft directory configuration	Visual Studio errors
26	0,00813	end gem call app[web actionpack rails block ruby c:/ruby vendor/bundle/ruby require lib/ruby/gems activesupport lib/ruby def rack method render error true	Ruby errors
27	0,46041	i'm code problem i've debug file application work run works app error time it's don't fine debugging working doesn't issue	General debugging
28	0,02166	file line import def python return print error true call class main traceback upload app files script recent debug webapp	Python errors
29	0,00568	error lnk referenced symbol unresolved external const function defined void public thiscall libcs nil cdecl end begin lpsolve j.o class	C/C++ errors

Table A.5: Topic modelling results with all Java posts and 30 topics (part 2).

Topic #	Weight	Key words	Possible descriptive label
0	0,18841	code debugging i'm memory debug time program find gdb question it's information i've don't good tool data java source linux	Java/GDB debugging tool on Linux
1	0,17183	debug debugging visual studio eclipse debugger code run application windows breakpoint running project start process window step set breakpoints i'm	Visual Studio/Eclipse debugging using breakpoints
2	0,14994	project file debug build files release error folder visual studio dll solution library application directory path run configuration projects mode	Visual Studio configuration errors
3	0,14055	server error web application i'm debug service database app problem page asp.net iis php i've data debugging file request working	Debugging ASP.NET/PHP web applications
4	0,05498	app xcode debug ios error iphone flash simulator device build application i'm run problem mac version fine code i've running	Xcode iOS debugging errors
5	0,08846	log file debug output script command run python line console test i'm logging error print tests running rails code messages	Scripting issues
6	0,12848	code class function string i'm object variable method line debug return values int debugging public debugger variables type array call	General debugging
7	0,10957	javascript i'm page code debug chrome i've problem browser work debugging file html works working image issue doesn't it's error	Debugging web applications in the browser
8	0,10318	exception error code application thread problem program call method line message debug debugger process exceptions i'm stack event run windows	General debugging
9	0,06544	android app device debug phone application eclipse emulator debugging windows usb adb google run i'm problem devices error key sdk	Eclipse Android debugging errors on Windows

Table A.6: Topic modelling results with all general posts and 10 topics.

Topic #	Weight	Key words	Possible descriptive label
0	0,11665	code debugger breakpoint debugging step debug line set breakpoints break source point hit stop program gdb function execution put i'm	GDB breakpoint management
1	0,06191	page request server debug django url user php response http app session post site requests rails set i'm email controller	Debugging HTTP requests
2	0,08356	thread process application running time threads task app run problem stop debug debugger debugging start loop call program background hangs	Debugging issues with multi-threading
3	0,03734	log logging file debug logs level messages net logger info git application message output set configuration write files console filter	Logging issues
4	0,1848	error problem message errors code found issue failed fine line debug unable find working run fix version works i'm fails	General debugging issues
5	0,03903	database data sql query server connection table mysql stored debug model entity queries sqlite user string procedure tables select application	SQL debugging
6	0,06424	eclipse java debug project run application plugin debugging jar intellij source netbeans ide tomcat mode running ant code maven idea	Debugging Java applications in the IDE
7	0,08506	button window click form debug event control application user menu dialog open view text code show windows tab box change	Debugging user interfaces
8	0,07243	web server service application asp.net iis mvc debug web.config site project page app website wcf running local net client development	Debugging .NET web applications
9	0,09945	file files folder path directory project copy source find application xml open created location create code access read load work	File I/O issues
10	0,05495	string code line xml i'm number array characters text loop output result character data strings end problem input values int	Issues with Strings/Arrays
11	0,02816	test tests run unit testing debug running delphi nunit coverage fails junit selenium fail runs resharper end i'm mstest runner	Unit testing issues
12	0,04767	function code debug int return foo define call include functions compiler mode program macro bar release defined gcc char void	General debugging issues
13	0,27534	code debugging i'm question good find i'd make don't time tool it's information lot tools i've development write specific things	General debugging/tooling issues
14	0,23153	i'm i've it's problem can't code work working doesn't find don't issue debugging debug found fine time edit works wrong	General debugging issues
15	0,03684	app android google key api debug application facebook apk map maps sdk signed play release manifest keystore store certificate file	Android Google/FB API authentication issues
16	0,04216	image screen game images video view problem i'm opengl layout works display background size debug fine color play code work	OpenGL debugging issues
17	0,0437	output console debug print window messages program application message statements write log text debugging display show i'm information file date	Logging issues
18	0,05404	android device app phone emulator usb debugging debug eclipse adb application devices windows run connect running connected logcat mobile sdk	Eclipse Android debugging issues
19	0,04916	app xcode ios iphone debug simulator device build run application mac ipad running project fine profile sdk version release crash	Xcode iOS debugging issues
20	0,05874	javascript chrome browser page debug html firefox script web jquery css debugging tools firebug developer work console files safari i'm	Debugging web applications in the browser
21	0,1327	visual studio windows application debug debugging run project net process program debugger start running bit dll machine installed exe attach	Debugging .NET applications in VS
22	0,0612	command script python run line gdb debug program file shell running perl arguments commands emacs import debugger set terminal scripts	Debugging scripts
23	0,0212	flash flex builder application spring air player debug swf tomcat mono component i'm version app monodevelop adobe debugger file actionscript	General debugging

Table A.7: Topic modelling results with all general posts and 30 topics (part 1).

24	0,11445	project build debug release library dll file solution projects configuration mode files studio compile visual assembly version set reference libraries	Visual Studio build configuration issues
25	0,06076	memory time debug slow performance size application takes kernel program linux cpu running usage heap seconds debugging data leak address	Debugging performance/memory issues
26	0,05927	class method public code void static call called object return private methods function null string classes var true constructor calling	Java/C# development
27	0,06266	exception stack code crash exceptions error trace application crashes catch thrown line program call debug report debugger throw information problem	General debugging
28	0,06866	object variable variables debugger values debugging list objects watch type i'm array view class window data property access set function	Watching variable values while debugging
29	0,04934	server php remote debug xdebug machine debugging running debugger port run windows local connect client node i'm installed linux netbeans	Remote debugging issues

Table A.8: Topic modelling results with all general posts and 30 topics (part 2).

Topic #	Weight	Key words
0	0,18487	debugger program error exception execution loop stop pause continue exceptions delphi point resume running thrown errors catch failed time times
1	0,19407	visual studio debugging project hit debug solution asp.net edit projects make page point continue time web option window change settings
2	0,10529	source code net enable debugging framework option disassembly options debug view disable i've enabled symbols window step checked tools stack
3	0,07406	python console print pdb log output pydev results trace text interactive def messages message i'd script action option logging django
4	0,24698	source file code files debug debugging open library can't window sources find compiled show project path build assembly inside location
5	0,08762	debugger assembly instruction i'm memory address windbg ollydbg mov instructions windows asm bit register eax don't process win ida command
6	0,19014	function call method stack called thread debugger trace functions i'm calls find return event current threads frame methods time ruby
7	0,06389	int main line class breakpoint string void public static result test return end step world loop private variable object simple
8	0,28053	step code debugger debugging i'm stepping method xcode steps skip function calls don't inside it's doesn't functions debug library press
9	0,06857	condition conditional statement debugger expression block true matlab case evaluate monodevelop slow add expressions java evaluation wait response debugbreak enter
10	0,12991	gdb program xcode debugging file command function symbols ios main lldb creator problem linux executable gcc library table make list
11	0,0378	error library set kernel shared device application target object cuda memory insert host gdb remote arm data machine external loading
12	0,14528	debug loaded symbols process dll hit attach project breakpoint service remote application debugging solution symbol file document server windows pdb
13	0,32867	line code lines debugging statement current debugger i'm executed number back it's execution problem jump find execute cursor feature executing
14	0,21734	break point points set debugging stop put running debugger application entry process breaks run breaking flash main working flow setting
15	0,74661	breakpoint breakpoints set debugger debug i'm stop hit app doesn't problem work run i've working can't don't setting works application
16	0,05286	netbeans php xdebug process debugging start controller request index.php child phpstorm variables output vim browser pdt plugin zend phpunit play
17	0,15889	eclipse debug java android debugging class step application found project view method source classes breakpoints line app cdt ide sdk
18	0,48292	code debug program debugging run mode variable time execution running change application variables find make execute edit executed values question
19	0,11624	javascript debugger script chrome test page code debugging firebug browser debug tests line unit tools jquery break execution firefox file

Table A.9: Topic modelling results with all general posts about breakpoints and 20 topics.

Topic #	Weight	Key words
0	0,28642	run eclipse project debug configuration launch configurations play click run/debug create button add application app option framework created file set
1	0,21848	error problem line issue exception version fine message java windows works errors i'm fix jre information bit command thread unable
2	0,08905	ant build command script run task build.xml line file target javac compile builder compiler tool flex shell generate builds custom
3	0,0577	java jni native version jvm dll library code linux environment jdk runtime build api memory call reference bit leiningen openjdk
4	0,05748	pydev javascript ide python debugger php file aptana ruby rails script html editor step support rhino import setup simple environment
5	0,7173	eclipse debug application java debugging code mode run running ide breakpoints start debugger breakpoint working program work question make doesn't
6	0,12439	eclipse cdt gdb program error windows run build ide c/c linux visual debug installed project studio path world cygwin make
7	0,12545	change scala time code hot make restart sbt files changed classes takes version development update class slow long save clean
8	0,12125	plugin eclipse installed plugins install plug-in update feature version create custom installation rcp extension development missing editor created software work
9	0,37951	i'm i've java it's don't set work can't doesn't working running find problem app found setting environment debugging test fine
10	0,25463	file jar project files java class folder eclipse directory path library main problem find classpath create works fine add program
11	0,11685	remote java application debugger applet debugging debug jvm process port running attach start connect server local browser connection remotely xdebug
12	0,06285	class method string classes public code import void static null swing called main methods function object package java generated add
13	0,19051	server tomcat web application jboss debug running deploy war app start jetty jsp mode spring glassfish eclipse deployed weblogic webapp
14	0,16098	source code eclipse debugging step sources found class file classes path find jdk debug can't edit attach files jar lookup
15	0,12619	project maven build projects eclipse dependencies workspace dependency plugin import repository mvn question install problem module goal building clean local
16	0,05895	test tests line junit command class launcher run unit jdb set breakpoint main visualvm default found true method testing attributes
17	0,1273	debug eclipse console output perspective view variable variables display values show log file window input shows text rcp back swt
18	0,11503	intellij idea gwt app mode run application debug project google web grails plugin development compile engine client i'm maven browser
19	0,1101	netbeans android project debug app emulator error blackberry device gradle adt file click apk ide tools ddms sdk development native

Table A.10: Topic modelling results with all general posts about Java IDEs and 20 topics.

Topic #	Weight	Key words
0	0,06192	map library stl vector data structure container struct size std::vector elements pointer std::string const std::map containers matrix file pointers order
1	0,14463	array values arrays int elements debugger debugging loop element i'm show index time creator contents byte program shows problem access
2	0,1099	string tostring type debugger strings enum display i'm i'd make representation custom date detail simple text double edit displayed override
3	0,17237	visual studio window debugger watch visualizer debugging display project data locals autoexp.dat type visualizers members show windows member net types
4	0,29311	error code i'm null type problem works exception debugger line i've issue fine message property it's var application working doesn't
5	0,5421	object objects i'm debugging debug code find access debugger don't can't properties instance it's class inspect method reference i'd i've
6	0,05938	json app server file service user data client request log string version windows control activity save workflow url web var
7	0,09632	list dictionary item key items keys loop i'm arraylist collection lists foreach added element elements empty create adding remove debugger
8	0,02488	treeview component source excel information ienumerable set call users vba tree data nsdata integer index interop understand tuple convert dictionary
9	0,06834	expression session evaluate lambda expressions evaluation page result asp.net context evaluated scala jsp unable net intellij mvc repl message hibernate
10	0,14653	data view debug tree xml results node table i've time collection app attributes query field database serialization nodes linq fields
11	0,08012	xcode debugger lldb print console nsstring ios gdb nil values description shows summary iphone nsdictionary breakpoint objective-c property nsmutablearray nslog
12	0,27437	variable variables function local global scope debug php defined var declared access it's question functions environment can't current code i've
13	0,16149	print gdb python contents debugging file dump list program i'm command structure names code question examine perl type content printing
14	0,10736	memory address pointer location reference release debug program pointers process null set references mode addresses application pointing crash heap initialized
15	0,48788	variable debugging variables values debug eclipse watch view debugger window code show set i'm change display mode breakpoint shows hover
16	0,16415	class type classes public field static private custom types method debug int default work void information question base member fields
17	0,06928	test copy instance controller method thread end unit statement ruby set tests false expected state undefined call understand multiple target
18	0,04631	property properties set find dynamic file guid read documentation i'm system prop column interface application working prototype spring version namespace
19	0,09452	function method return foo call arguments pass returns reference parameter path bar methods parameters result values code string passed called

Table A.11: Topic modelling results with all general posts about watches and 20 topics.

Appendix B

Online Survey - Printed Version

A printed version of our online survey on the perception of debugging can be found on the next page.

The Perception of Debugging

This survey investigates how software developers debug. Its results will be compared to TestRoots WatchDog data [1].

It takes 5 minutes to complete and is hassle-free. By entering your e-mail address at the end of the survey you can win one of the three €15 Amazon vouchers.

Niels Spruit
MSc Computer Science
Delft University of Technology
n.spruit@student.tudelft.nl
twitter.com/n_spruit

[1] http://testroots.org/testroots_watchdog.html

*Required

Background information

1. **Experience in software development** *
   Mark only one oval.
   - ◯ <1 year
   - ◯ 1-2 years
   - ◯ 3-6 years
   - ◯ 7-10 years
   - ◯ >10 years

2. **Programming languages you use** *
Tick all that apply.

- [ ] Java
- [ ] Python
- [ ] PHP
- [ ] C#
- [ ] C++
- [ ] C
- [ ] JavaScript
- [ ] Objective-C
- [ ] Swift
- [ ] R
- [ ] Other: .....

3. **Integrated Development Environment (IDE) you use or like most, which we will call 'your IDE' from now on** *
Mark only one oval.

- ◯ Eclipse
- ◯ Visual Studio
- ◯ Vim
- ◯ Xcode
- ◯ NetBeans
- ◯ Sublime Text
- ◯ IntelliJ
- ◯ Komodo
- ◯ Xamarin
- ◯ Emacs
- ◯ Other: .....

IDE-provided debugging infrastructure (1/3)

4. **Do you use the debugging infrastructure provided by your IDE?** *
Mark only one oval.

- ◯ Yes *After the last question in this section, skip to question 7.*
- ◯ No *After the last question in this section, skip to question 6.*
- ◯ My IDE does not have a debugger. *After the last question in this section, skip to question 10.*

5. **For debugging ...** *
   Tick all that apply.

   - [ ] I use an external program (e.g. GDB).
   - [ ] I use the IDE debugger.
   - [ ] I use print statements.
   - [ ] I examine the log files.
   - [ ] I use additional other techniques.
   - [ ] none of the above applies.
   - [ ] Other: .....

IDE-provided debugging infrastructure (2/3)

6. **I do not debug in the IDE, because ...** *
   Tick all that apply.

   - [ ] I use an external program that I find more effective/efficient.
   - [ ] print statements are more effective/efficient.
   - [ ] techniques other than print statements are more effective/efficient.
   - [ ] I don't know how to use it.
   - [ ] my program is impossible to debug.
   - [ ] I don't debug my programs.
   - [ ] Other: .....

Skip to question 10.

IDE-provided debugging infrastructure (3/3)

The purpose of this section is to get to know which of the debugging features provided by many IDEs you (don't) know. Moreover, this section aims to answer which of these features are actually used by developers in practice.

7. **Breakpoint types** *
   Mark only one oval per row.

	I don't know	I know	I know and I use
Line breakpoint	◯	◯	◯
Temporary line breakpoint	◯	◯	◯
Class prepare breakpoint	◯	◯	◯
Exception breakpoint	◯	◯	◯
Method breakpoint	◯	◯	◯
Field watchpoint	◯	◯	◯

8. **Breakpoint options** *
   Mark only one oval per row.

	I don't know	I know	I know and I use
Specifying a condition			
Specifying a hit/pass count			
Setting the suspend policy			

9. **Other debugging features** *
   Mark only one oval per row.

	I don't know	I know	I know and I use
Stepping through the code			
Inspecting variable values			
Inspecting the call stack			
Defining watches			
Evaluating expressions			
Modifying variable values at runtime			
Editing code at runtime (hot swapping)			

Debugging using unit tests

10. **Do you write small unit tests to be able to reproduce bugs and start debugging?** *
    Mark only one oval.

    - Yes
    - No

11. **Do you use unit tests to check your progress during the debugging process?** *
    Mark only one oval.

    - Yes
    - No

12. **Do you use unit tests to verify the correctness of a possible bug fix?** *
    Mark only one oval.

    - Yes
    - No

Final remarks

13. **"The best invention in debugging still was printf debugging." What is your opinion? If you want to share something else on debugging, feel free to use the answer field below.**

.....

.....

.....

.....

.....

14. **If you are interested in winning a €15 voucher for Amazon, enter your e-mail address below. We might contact you for a (non-obligatory) follow-up interview.**

.....